FACIAL KINSHIP VERIFICATION WITH LARGE AGE VARIATION USING DEEP LINEAR METRIC LEARNING

DIEGO DE OLIVEIRA LELIS

DISSERTAÇÃO DE MESTRADO EM SISTEMAS MECATRÔNICOS
DEPARTAMENTO DE ENGENHARIA MECÂNICA

FACULDADE DE TECNOLOGIA

UNIVERSIDADE DE BRASÍLIA

UNIVERSIDADE DE BRASÍLIA
FACULDADE DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA MECÂNICA

FACIAL KINSHIP VERIFICATION WITH LARGE AGE VARIATION USING DEEP LINEAR METRIC LEARNING

DIEGO DE OLIVEIRA LELIS

Orientador: PROF. DR. DÍBIO LEANDRO BORGES, CIC/UNB

DISSERTAÇÃO DE MESTRADO EM SISTEMAS MECATRÔNICOS

PUBLICAÇÃO PPMEC - YYY/AAAA
BRASÍLIA-DF, 06 DE DEZEMBRO DE 2018.

UNIVERSIDADE DE BRASÍLIA
FACULDADE DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA MECÂNICA

FACIAL KINSHIP VERIFICATION WITH LARGE AGE VARIATION USING DEEP LINEAR METRIC LEARNING

DIEGO DE OLIVEIRA LELIS

DISSERTAÇÃO DE MESTRADO ACADÊMICO SUBMETIDA AO DEPARTAMENTO DE ENGENHARIA MECÂNICA DA FACULDADE DE TECNOLOGIA DA UNIVERSIDADE DE BRASÍLIA, COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM SISTEMAS MECATRÔNICOS.

APROVADA POR:

Prof. Dr. Díbio Leandro Borges, CIC/UnB
Orientador

Prof. Flávio de Barros Vidal, CIC/UnB
Examinador interno

Prof. George Luiz Medeiros Teodoro , CIC/UnB
Examinador externo

BRASÍLIA, 06 DE DEZEMBRO DE 2018.

FICHA CATALOGRÁFICA

DIEGO DE OLIVEIRA LELIS
Facial Kinship Verification with Large Age Variation using Deep Linear Metric Learning
2018xv, 82p., 201x297 mm
(ENM/FT/UnB, Mestre, Sistemas Mecatrônicos, 2018)
Dissertação de Mestrado - Universidade de Brasília
Faculdade de Tecnologia - Departamento de Engenharia Mecânica

REFERÊNCIA BIBLIOGRÁFICA

DIEGO DE OLIVEIRA LELIS (2018) Facial Kinship Verification with Large Age Variation using Deep Linear Metric Learning. Dissertação de Mestrado em Sistemas Mecatrônicos, Publicação yyy/AAAA, Departamento de Engenharia Mecânica, Universidade de Brasília, Brasília, DF, 82p.

CESSÃO DE DIREITOS

AUTOR: Diego de Oliveira Lelis
TÍTULO: Facial Kinship Verification with Large Age Variation using Deep Linear Metric Learning.
GRAU: Mestre ANO: 2018

Diego de Oliveira Lelis
diego.o.lelis@gmail.com

Acknowledgments

I would like to say thanks to my Master's degree advisor, Prof. Dr. Díbio Leandro Borges, for all the guidance an support throughout this research, to the University of Brasília(UnB) that provided me with an exceptional environment for research and learning.

I would also like to extend my thank you to all the professors that enriched my knowledge on this journey: Prof. Dr. Li Weigang, Prof. Dr. José Maurício S. T. da Motta, and Prof. Dr. Andrea Cristina dos Santos.

Abstract

Facial appearance affects how humans interact. It is how relatives are visually identified to determine how social interactions proceed. Humans can identify kin relations based only on the face. Intrinsically, giving the ability to detect kin relations to computers can improve their usefulness in our daily lives. This research proposed a solution to the kinship verification problem with a novel non-context-aware approach using a dataset with large age variation by applying our proposed method Deep Linear Metric Learning(DLML). Our method leverages multiple deep learning architectures trained with massive facial datasets. The knowledge acquired on traditional facial recognition tasks is re-purposed to feed a linear metric learning model. The proposed method was able to achieve better performance than other context-aware methods on tests that are inherently more difficult than the ones used on previous methods with the UB Kinface dataset. The results show that our method can use the knowledge of deep learning architectures trained to perform mainstream facial recognition tasks with massive datasets to solve kinship verification on the UB Kinface database with robustness towards large age differences present on the dataset. Our method also offers enhanced applicability when compared to previous methods on real-world situations, because it removes the necessity of knowing/detecting and treating large age variations to perform kinship verification.

SUMARY

LIST OF FIGURES

LIST OF TABLES

LIST OF SOURCE CODES

LIST OF TERMS AND ACRONYMS

ANN	Artificial Neural Networks
ConvNets	Convolutional Neural Networks
CUDA	Compute Unified Device Architecture
DLML	Deep Linear Metric Learning
DMML	Discriminative Multimetric Learning for Kinship Verification
fcDBN	Filtered Contractive Deep Belief Network
LFW	Labeled Faces on The Wild
MNRML	Multiview Neighborhood Repulsed Metric Learning for Kinship Verification
MTCNN	Multi-Task Convolutional Neural Network
P-Net	Proposal Network
PDFL	Prototype-Based Discriminative Feature Learning for Kinship Verification
R-Net	Refine Network
ReLU	Rectified Linear Unit
TL	Self-Similarity Representation of Weber Faces for Kinship Classification
TL	Transfer Learning
TSL	Transfer Subspace Learning
Visual Attr.	Visual Attributes

Chapter 1

Introduction

Different from the most common facial recognition approaches that mostly try to compare similarity, kinship verification is more complicated to solve because people with dissimilar appearances can be kin and people with similar appearances can be non-kin at all [Shao et al. 2011] [Georgopoulos et al. 2018] [Kohli et al. 2017] [Lu et al. 2014]. For instance, on Figure 1.1, pairs a-b and c-d are non-kin similar people, this proximity between facial characteristics provides a complex challenge for facial recognition models because it is necessary to identify what features can signal a kin relation to avoid false positives like the ones that it could easily occur between pairs a-b and c-d, for example.

Despite the difficulties to perform kinship verification, humans can identify kinship relations at a higher rate than chance, but it is not clear how [Dehghan et al. 2014]. In this research is also added the additional factor of large age variations with the UB Kinface dataset [Shao et al. 2011].

Since the old parent's face structure is transformed when compared to when they were young [Shao et al. 2011], the age difference increases the distance between the face of child-old parent making it more difficult to identify the kin relation. On Figure 1.2, it is possible to observe two examples of pairs of images (a-b and c-d) that because of the large age differences it would easily prompt a false negative if the model is based solely on the raw facial distance. The age difference present on the UB Kinface dataset makes the problem more challenging [Shao et al. 2011], and it has been treated separately by previous methods available on the literature [Georgopoulos et al. 2018] [Kohli et al. 2017] [Xia 2012] [Yan et al. 2014] [Yan et al. 2015] [Lu et al. 2014] [Xia et al. 2011] [Shao et al. 2011] [Kohli et al. 2012].

A key factor that inspired this research is the fact that all the other solutions for kinship verification with large age variations using the UB KinFace database, either try to preprocess the face of the old parent to approximate it to the child face as shown in Figure 1.3, or trained the same method twice, one for the child-young parent pairs, and another for child-old parents pairs like on Figure 1.4.

Our complete DLML proposed method is displayed on Figure 1.6. The method is divided

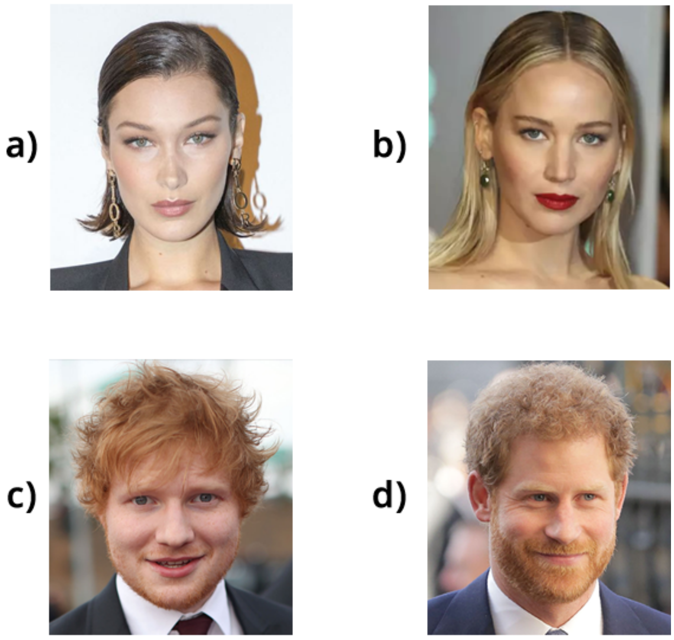


Figure 1.1: Images of similar non-kin people, a) to b), and c) to d) - images obtained from: https://goo.gl/qsgRFU, https://goo.gl/6tnHkN, https://goo.gl/wAhHv8.

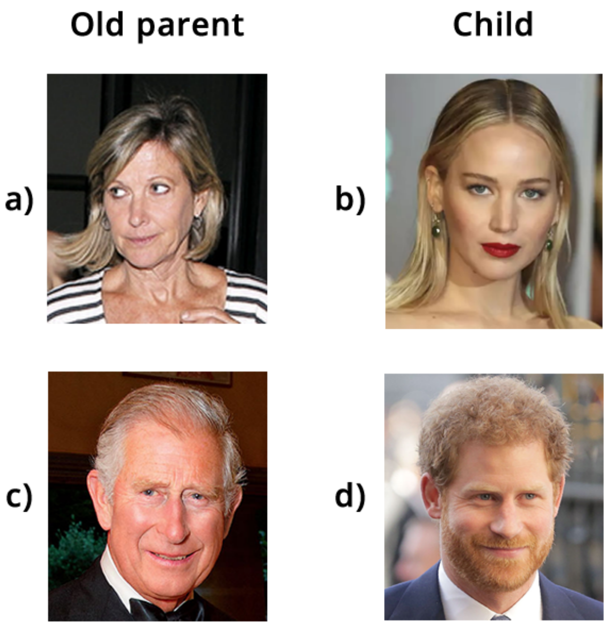


Figure 1.2: Images of kin people with large age difference- images obtained from: https://goo.gl/6tnHkN, https://goo.gl/wAhHv8, https://goo.gl/AE3E4d, https://goo.gl/fpU3Cj.

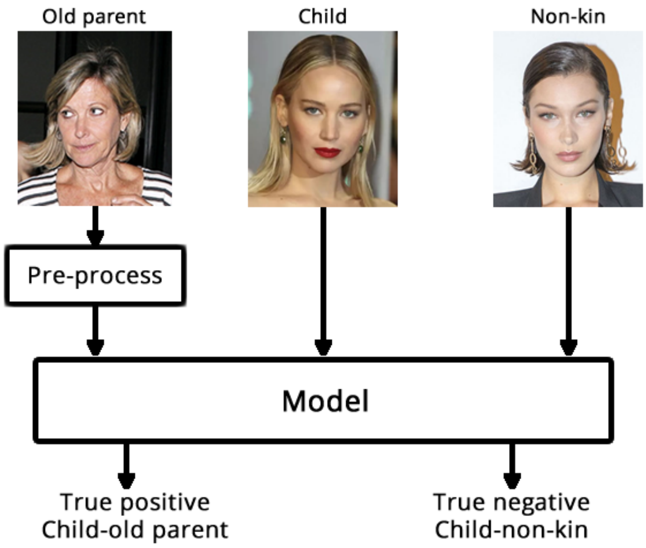


Figure 1.3: Approach of previous methods that approximate the old parents face from the child face by reducing aging effects. Desired outputs are showed at last stage.

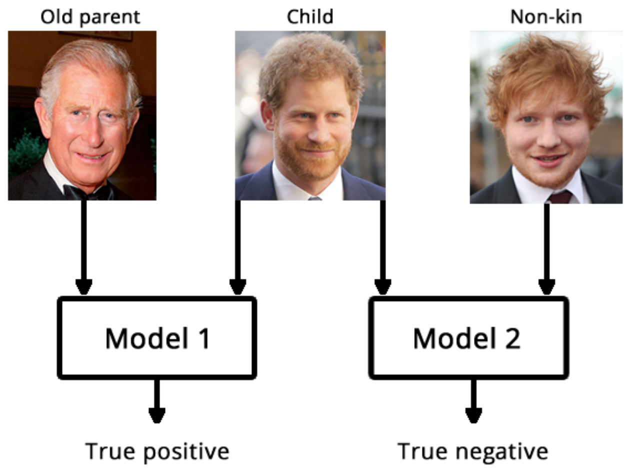


Figure 1.4: Approach of previous methods that trained and tested the same method separately for child-young parent and child-old parent pairs with desired outputs.

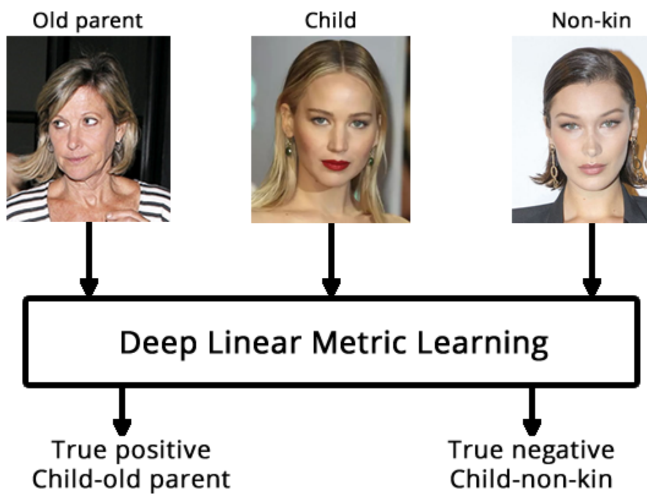


Figure 1.5: Approach of of our method that it does not treat differently child-young parent and child-old parents pairs.

into four stages:

- **Face Alignment-MTCNN:** Faces are detected and cropped using a Multi-Task Convolutional Neural Network(MTCNN) [Zhang et al. 2016]. The first phase will provide a picture of the face with 160x160 size as output.
- **Feature Extraction-FaceNet:** The processed images are then fed onto a FaceNet [Schroff et al. 2015] [Sandberg 2018] implementation that is going to generate embeddings of 128 dimensions of the face.
- **Feature subtraction:** The extracted features are subtracted to create an array of 128 dimensions that represents the distance between two faces.
- **Linear model:** Finally, the distance array of 128 dimensions is fed onto a linear model that is going to provide a boolean output informing if the two people are kin or not.

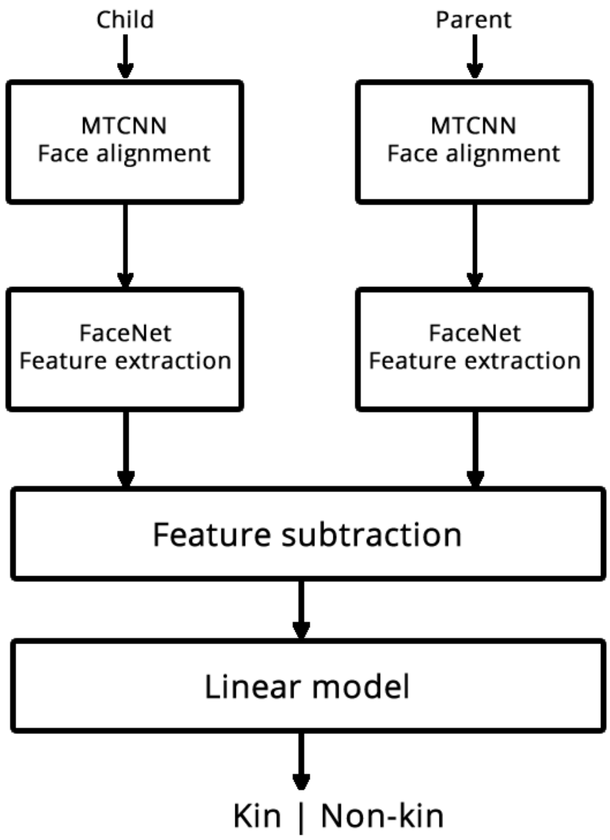


Figure 1.6: The complete proposed method to perform kinship verification.

1.1 Applications

Among the applications of kinship verification it is possible to cite:

- On passports checks because it is necessary to differentiate kin people. Kinship verification can be used to improve facial recognition models that are sensitive to this type of situation [Georgopoulos et al. 2018].
- Identifying the parents of lost children and orphans to help the work of law enforcement agencies [Lu et al. 2014].
- Improving target ads by using the preferences of their kin people to provide a more personalized experience [Georgopoulos et al. 2018].
- To organize family photos detecting kin relations on pictures.
- To search for relatives in public datasets [Kohli et al. 2017].
- To allow make-up artists to modify the appearance of two people in a way that they seem blood-related [Georgopoulos et al. 2018].

Our method offers a more practical and simple solution to all of these applications because it discards the need for detecting large age differences.

1.2 Contributions

The contributions of this research are:

- **Deep Linear Metric Learning:** Until now, all the past solutions have treated kinship verification with large age variations using the UB Kinface dataset as two separate problems, identify a child-young parent kin relation, and identify a kin relation between child-old parent. Our novel DLML method offers a new and more practical solution for kinship verification problem with large age variations, by using an all in one approach that enhances applicability on real-world situations.
- **Transfer learning:** The results confirmed that the features extracted by our FaceNet model trained with VGGFace2 to perform facial recognition can be re-purposed to perform kinship verification with robustness towards large age variations present on the UB Kinface dataset by applying our linear metric learning approach.
- **Results:** The results provided by this research showed that the proposed DLML framework can identify kinship relations despite large age differences and with better performance than multiple other methods.

1.3 Final considerations

On this chapter, the kinship verification with large age variation problem tackled by this research was presented and explained why it is a difficult problem, the components of the proposed DLML method were explained in a high level. A few applications and the main objectives of the research were presented. The contributions of the research are also cited at the end of the chapter.

It is important to highlight that this research does not have the purpose of discussing/exploring how the kinship relation is detected, only to perform the task. The reason for that is because as stated by [Dehghan et al. 2014], it is not clear how humans can identify kinship relations, because of that, trying to understand how these process works are usually treated as a different type of research. Exploring why the kinship relation exists is an interesting next step for this research, but it is something that on this moment has not yet explored.

Chapter 2

Background

This chapter will present in high level the main resources used on this research.

2.1 The Artificial Neuron

All of our architectures are based on the first artificial neuron model was presented at the decade of 1940, [McCulloch and Pitts 1943], even today this is one of the most used models on artificial intelligence [Goodfellow et al. 2016]. It is inspired by the brain and tries to mimic how human neurons are activated [Rosenblatt 1958]. It can also be called the neuron element [Widrow and Hoff 1960].

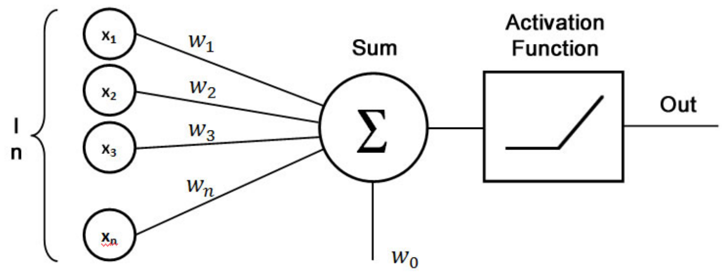


Figure 2.1: The artificial neuron

$$y = \sum_{i=1}^{j} w_i \times x_i + w_0 \tag{2.1}$$

As shown at the Equation 2.1 the artificial neuron receives multiple inputs($x_1, x_2...x_n$), each input is multiplied by a correspondent weight($w_1, w_2...w_n$), the results are summed up; a bias(w_0) is added to improve the freedom of the model; the outcome of this sum (y) act as an input to an activation function that will decide if the artificial neuron should be activated or not [Rosenblatt 1958].

2.1.1 Rectified Linear Unit

Our FaceNet and MTCNN models use one of the most successful activation functions, the Rectified Linear Unit (ReLU) [Goodfellow et al. 2016]. This function is heavily inspired by how neurons work. As presented on the Equation 2.2, the output of the artificial neuron is going to be the maximum value between the sum of inputs and weights (y) and 0 [Goodfellow et al. 2016]. At the Figure 2.2, it is possible to observe that the (w_0) bias is going to set the point of activation of the output, where the output starts to increase accordingly to the input stimulus.

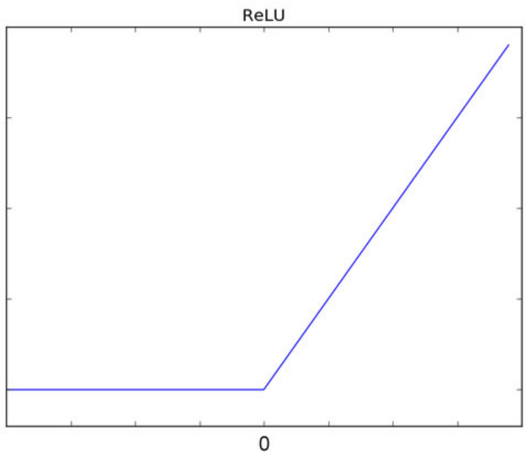


Figure 2.2: ReLU - Rectified Linear Unit Activation Function

$$Out = max(0, y) \tag{2.2}$$

2.1.2 Leaky Rectified Linear Unit

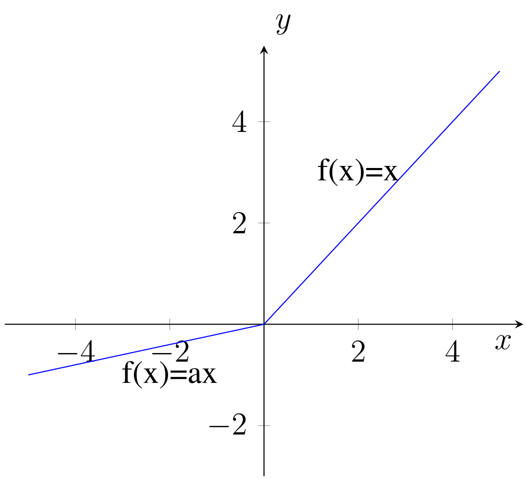


Figure 2.3: Leaky ReLU function: Before activation the information the function is defined by:$f(x) = ax$, and after: $f(x) = x$

Our linear metric learning model activation function is a variation of the ReLU, the Leaky Rectified Linear Unit (Leaky ReLU) [Xu et al. 2015] is used on the last stage of the model on Figure 1.6, and it is very similar to the ReLU activation function, the only difference is that before the activation the output is defined by a function instead of zero as shown in Figure 2.3. By using a different function before activation loss of information is avoided while the output is not active.

2.1.3 Training

One of the main tasks on the artificial neuron model is to find the best value for the weights that will ensure the right output; this process is called training. On training, the artificial neuron receives inputs that have the desired outputs informed. Each time that the output is wrong, the error is calculated by a given function, and the weights are adjusted [Goodfellow et al. 2016] [Rosenblatt 1958] [Widrow and Hoff 1960]. This strategy of training is called supervised learning, and after finished, the artificial neuron can operate without having the desired output informed [Goodfellow et al. 2016].

2.1.4 Loss

Loss functions used on training to measure the performance of the prediciton, saying how far the answer is from the desired [Goodfellow et al. 2016]. The values provided by this function will be used on an otimization function in order to try to find the minimum value of the loss function by adjusting the weights. The most common optimization method function for multi-layer ANN's is called backpropagation.

2.2 Artificial Neural Networks-(ANN)

The artificial neuron is a feed-forward model that works as a linear classifier, and because of that, it can only learn simple tasks. It can find out how to mimic an AND function, but it can not learn how to classify one image. To overcome that problem researchers connected multiple layers of artificial neurons(Figure 2.4. That way it is possible to solve complicated problems like image classification [Goodfellow et al. 2016]. When working with images, the inputs(i_0, i_1, i_2, i_3, i_4) would usually refer to a value between 0 and 255 if the data is black and white, or a vector of three values between 0 and 255 if the image has color information.

With the addition of more layers, the training process is more complicated. Identify what contribution one intermediate/hidden layer has to the error on the output becomes a challenge. It is necessary to understand what is the role of that layer in the whole process. The most efficient way to do that task is with a technic called backpropagation [LeCun et al. 1989] [Goodfellow et al. 2016]. Since the outcome of one layer is a func-

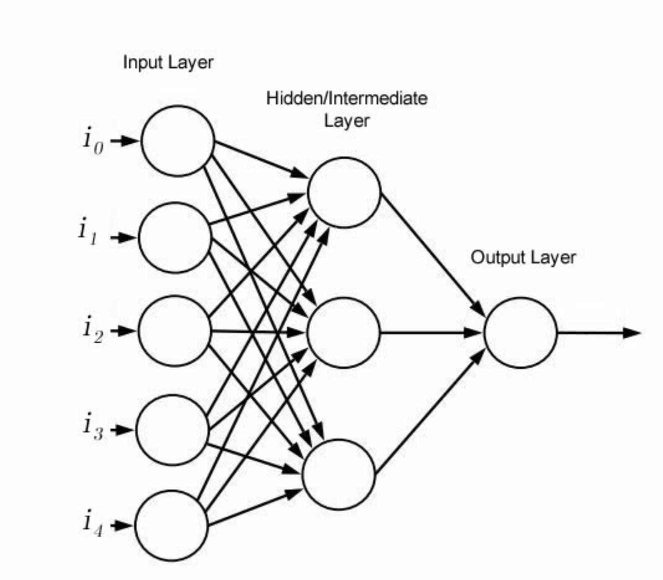


Figure 2.4: ANN - Artificial Neural Network

tion of the input of the previous layer, it is conceivable to use a cost function that represents the error and using the derivatives of this function to discover how much each layer contributes to the error of the output. After this information is acquired, it is possible to adjust the weights of each layer to improve the result. When working with images, the ANN learns how to extract features of the data on this process [LeCun et al. 2010].

2.3 Backpropagation

As the name suggests, the backpropagation algorithm is a technic to propagate errors back. The method treats each layer of the network as an independent function, and by assuming that, it is possible to use the chain rule of derivatives in order to understand how each layer is responsible for the error on the last layer [Goodfellow et al. 2016]. One of the most common methods for backpropagation is called gradient descent, this method tries to find the minimum of the loss function. There are also multiple variations of this method, and on this paper the standard backpropagation and the Adam [Kingma and Ba 2015] version are used.

2.4 Deep Learning

The first studies on artificial intelligence solved problems describing a list of formal mathematical rules. That is also known as the classical approach to artificial intelligence. That is exceptional for situations that it is possible to model your problem in mathematical rules, but not useful when dealing with problems that require intuitive knowledge such as face recognition and other computer vision tasks [Goodfellow et al. 2016].

Deep learning is a category of machine learning that uses artificial neural networks with many layers of artificial neurons, thus the name deep learning. Because of their depth, these models can solve problems that require intuitive knowledge. With that, it is possible to learn complex concepts without the necessity of model them into mathematical rules. Intermediate layers can be trained efficiently using backpropagation algorithms [LeCun et al. 2010]. Deep learning models can also be called the modern approach to artificial intelligence [Goodfellow et al. 2016].

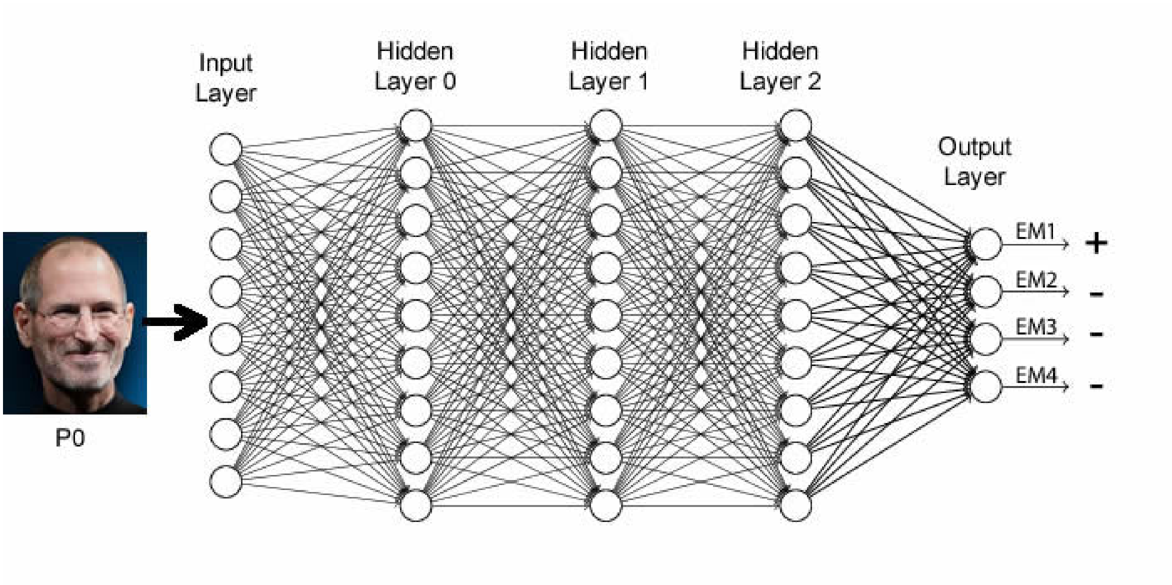


Figure 2.5: Deep neural network performing face classification - source: [Guo et al. 2016]

Figure 2.5 illustrates a trained deep neural network performing face recognition, where the face of the person of the class P0 is submitted to the system. The system will process the data using three hidden layers, and one output layer, to then give a positive result for the P0 output and a negative result for the remaining classes(P1, P2, P3).

With the recent advances in deep learning, computers were able to provide results that are greater than a person in computer vision tasks such as face recognition and image classification [LeCun et al. 2010].

2.5 Convolutional Neural Networks-(ConvNets)

Convolutional neural networks are one of the most successful artificial neural network architectures for feature extraction. These networks are inspired by the neocognitron [LeCun et al. 2010] [Goodfellow et al. 2016], a model based on the human visual cortex [Fukushima 1980].

Since proposed [LeCun et al. 1989], ConvNets have won major computer vision competitions. The state-of-the-art classification algorithm with the best result on the ImageNet Large-Scale Visual Recognition Challenge it is based on a convolutional architecture and has reached an error of 3.6% [Goodfellow et al. 2016]. The winner architecture of the ImageNet 2014, the inception [Szegedy et al. 2016], is used on this research.

One of the main advantages of ConvNets is the share of weights. After defining the size of the feature extraction region(Figure 2.6), the same weights will be used to extract the features of the input. The weight sharing improves the performance, because the training process becomes more straightforward, and the portion of memory used to store the weights are significantly smaller than the portion used by other architectures [LeCun et al. 1989] [LeCun et al. 2010] [Goodfellow et al. 2016]. Usually, after one or multiple convolutional layers, a pooling layer is used to reduce the dimensionality of the data minimizing information loss for the next layer as presented on Figure 2.6 [Goodfellow et al. 2016].

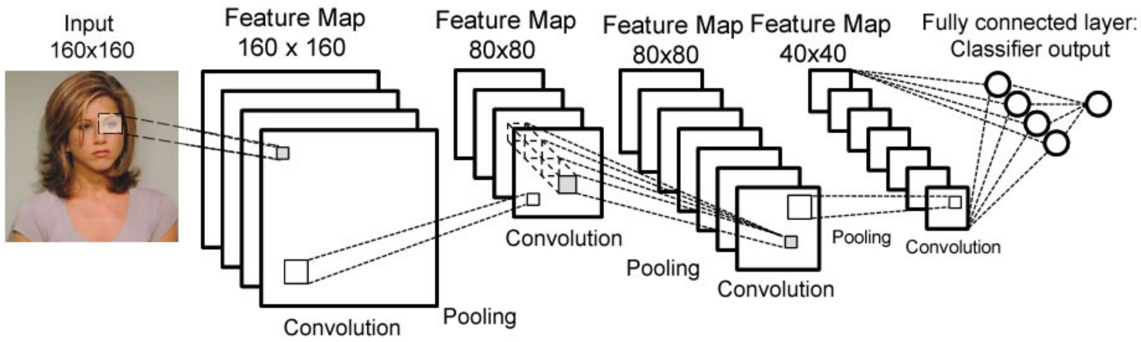


Figure 2.6: Convolutional Neural Network - source: [Guo et al. 2016]

2.6 Transfer Learning

Knowledge transfer is something that humans use to learn new complex concepts quickly; They can use knowledge acquired from other experiences to help them understand new representations and features of the world [Gutstein et al. 2008](Figure 2.7. ConvNets can also use the knowledge from one task to learn other tasks faster like humans do [Ranjan et al. 2016]. The most common way to do that with machine learning algorithms, including deep learning models, is to use the weights of an ANN or other characteristics of the model. These trained weights have the abstract representation of the input data to perform the feature extraction from the data.

When working with an ANN to utilize the knowledge acquired on previous tasks, it is possible to train the last layer, what will define if the knowledge can be used for the task at hand, is how much training is necessary to achieve good results. With image related tasks, researches have shown that the cost of retraining a model is small [Ranjan et al. 2016].

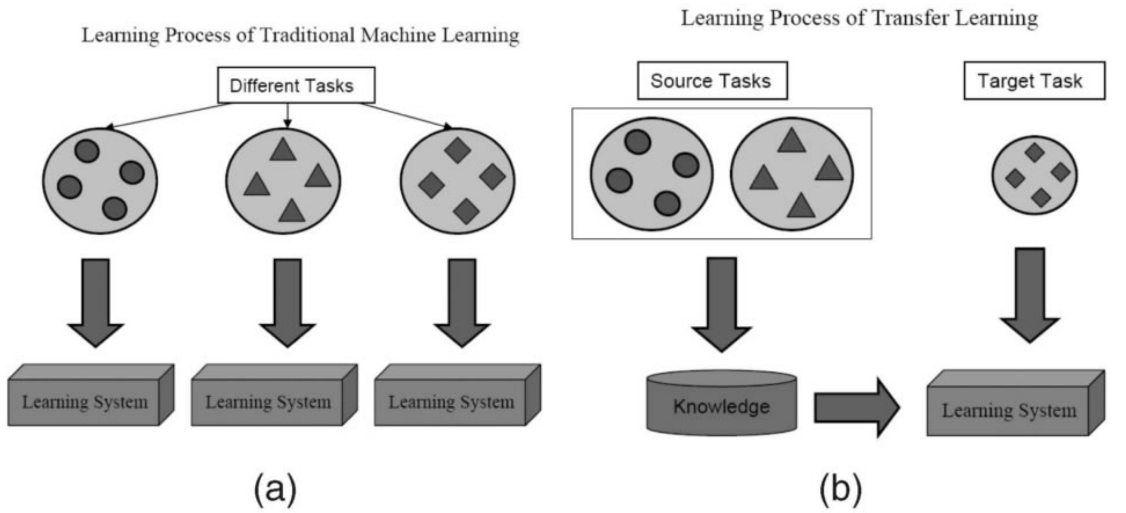


Figure 2.7: Different learning processes between (a) traditional machine learning and (b) transfer learning - source: [Pan and Yang 2010]

2.7 Inception Module

The first version of the inception architecture that is used on the original FaceNet is built with the inception module: a mix of layers that run several parallel convolutional layers and concatenate their outputs as presented on Figure 2.8.

The main idea of the inception module is to discover how the best local sparse structure in a convolutional network can be approximated and covered by readily available dense components [Szegedy et al. 2015]. The main benefit of this strategy is the increase of units at each stage without unconstrained computational complexity increase [Szegedy et al. 2015].

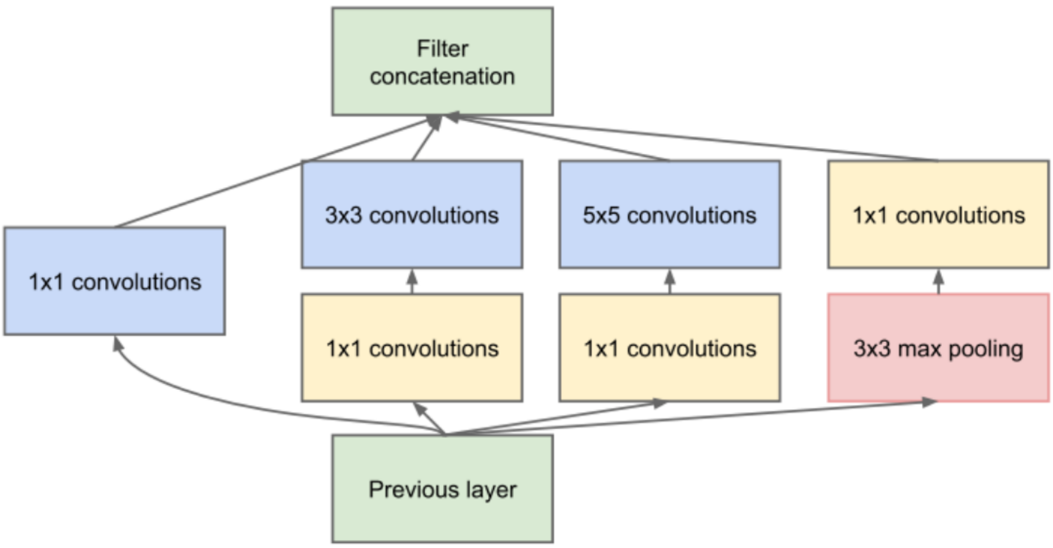


Figure 2.8: The Inception module - source: [Szegedy et al. 2015]

2.8 Final considerations

On this chapter the main necessary resources to understand this research were presented, informing the reader about the main necessary aspects to understand this research. These resources are artificial neuron, ReLU activation function, Leaky ReLU activation function, the training process for artificial neurons, ANN's, Deep Learning, ConvNets, Transfer Learning, Inception Module, and Inception-ResNet-v1 modules.

Chapter 3

Related Works

Making kin annotations is more complicated than making annotations of identity because it is necessary to work with pairs. Inherently, it is more challenging to collect and annotate the data of the UB Kinface than the data of VGGFace2 that it is mainly used to detect identity. This complexity led to a scarcity in large kin-related datasets when compared to traditional datasets such as Labeled Faces on the Wild (LFW) and VGGFace2 [Georgopoulos et al. 2018].

There is a consensus that the UB Kinface dataset is the kinship dataset with the largest age variations [Georgopoulos et al. 2018]; however, the original paper [Shao et al. 2011] does not provide the values of the age differences among pairs.

All the past solutions that used UB Kinface have focused mainly on achieving good results on the dataset, treating the child-young parent and child-old parent pairs as different problems [Georgopoulos et al. 2018](Figure 1.3 and 1.4. The methods found in the literature are difficult to apply in a real-world environment because they need to detect if there is a big age difference between the two faces to decide what approach should be used.

Table 3.1 presents some of the most relevant methods evaluated on the UB-kinface dataset [Georgopoulos et al. 2018]. In Table 3.1, the different models strategy refers to the approach showed on Figure 1.4, and pre-process refers to the approach presented on Figure 1.3, the 5-fold and leave-one-out columns on Table 3.1 are the average accuracy of these methods child-young parents and child-old parents, unlike our method these evaluations are performed separately.

Most of the attempts to solve kinship verification have used shallow machine learning methods like [Chergui et al. 2018], [Dehghan et al. 2014], [Yan et al. 2014], [Yan et al. 2015], [Xia et al. 2011], [Shao et al. 2011], [Kohli et al. 2012], [Xia 2012], [Lu et al. 2014]. The only deep learning method evaluated on the UB Kinface dataset used about 600,000 images for train on feature extraction [Kohli et al. 2017], more than five times less than our method that used more than three million images from VGGFace2 as shown on Table 5.1).

Table 3.1: Results of other methods on the UB Kinface dataset

Method	5-fold	Strategy
fcDBN [Kohli et al. 2017]	91.75%	Different models
Visual Attr. [Xia 2012]	82.50%	Only child-old par.
DMML [Yan et al. 2014]	72.25%	Different models
PDFL [Yan et al. 2015]	67.30%	Different models
MNRML [Lu et al. 2014]	67.05%	Different models
TL [Xia et al. 2011]	60.00%	Pre-process
TSL [Shao et al. 2011]	56.50%	Pre-process
SSRW [Kohli et al. 2012]	53.90%	Different models

One of the few deep learning methods available on literature that performed kinship verification on the UB Kinface dataset is called Filtered Contractive Deep Belief Network (fcDBN) [Kohli et al. 2017]. fcDBN is also , to the best of our knowledge, the state of the art method for most of the publicly available datasets, this method used for the first time external datasets to teach the model how to extract facial features to perform kinship verification [Georgopoulos et al. 2018].

In the first stage of fcDBN, the features of each facial region are learned from outside training data. These are learned through the filtered contractive DBN (fcDBN) approach. The learned representations are combined in a compact representation of the face in the second stage. Finally, a multi-layer neural network is trained using these learned feature representations for supervised classification of kin and non-kin [Kohli et al. 2017].

fcDBN was tested on the UB Kinface dataset using five-fold cross-validation and achieved 92.00% of accuracy on child-young parents pairs and 91.50% accuracy on child-old parents pairs [Kohli et al. 2017], 91.75% on average.

The research responsible for publishing the UB Kinface dataset [Shao et al. 2011] used a method called Transfer Subspace Learning(TSL) that uses local Gabor filters to extract features. These features are used to determine if parent and children have similar eyes, noses or mouths. With the extracted key points, six ratios of common regions distances are obtained, e.g., eye-to-eye versus eye-nose distance. The TSL method performs context-aware tests reducing the divergence between child-old parent by using the child-young parent as an intermediate set.

The research presented in [Shao et al. 2011] extracted the features using Gabor filters on each local region. These features are used to determine if parent and children have similar eyes, noses or mouths. With the extracted key points, six ratios of common regions distances are extracted e.g: eye-to-eye versus eye-nose distance. Following a principle that says that these distances are inherited mainly from parents. Structural information is also extracted and following a principle that says that old parent's structural face is transformed from the one when they were young, because of that a transfer subspace learning method was applied to mitigate the degrading factor. To fully utilize all features, a new strategy called Cumulative Match Characteristic (CMC) is used: features are added in several rounds according to the one that can maximize the difference of recognition performance of child-old parents. After extracting and select the features that will be used for classification a metric learning approach is used, this process will try to use the selected features to choose what features should determine if the two people are kin or not. On their research, they also presented two human baselines for kinship verification that are 53.17% and 56.00%. With 5-fold cross-validation, with 40 positive pairs and 40 negative pairs being left to test the accuracy result 56.5%. With leave-one-out protocol, the achieved accuracy was 69.67% [Shao et al. 2011]. The human baseline shows that the problem tackled by this research is a difficult one.

Another example of research on kinship verification is [Lu et al. 2014]; this paper proposed a novel model called Neighborhood Resulsed Metric Learning(NRML). In this case, the relations were separated into four different types: father-son (F-S), father-daughter (F-D), mother- son (M-S), and mother-daughter (M-D) kinship relations. The strategy used by this method tries to repulse interclass samples (without kinship relation) with the higher similarity that lie in a neighbourhood and approximate the intraclass samples, using the more discriminative information for solve the problem. They also used multiple feature descriptors to try to improve the performance of the method, in this case, it was called Neighborhood Resulsed Metric Learning(MNRML)

Our method differs from fcDBN on the architectures used (MTCNN and FaceNet), on the dataset used to train the network how to extract facial features (VGGFace2). However, the main difference between our DLML and all the previous methods including fcDBN is the fact that our method can detect kin relations on the UB Kinface dataset without treating any age difference, offering enhanced applicability.

Metric learning is used because is one of the most sucessfull methods to solve the kinship verification problem, since it does not need as many data as deep learning methods, and it can separate what features are relevant to detect a kin relation [Georgopoulos et al. 2018].

Furthermore, in our case, to show how expressive the extracted features of our method are to perform kinship verification on the UB Kinface, our last stage is a simple linear artificial neural network.

3.1 Final considerations

In this chapter, the method from the original UB Kinface dataset is presented [Shao et al. 2011], other methods evaluated on the UB Kinface dataset are also presented and compared to the method proposed in this research. Another important topic approached on this chapter is the state of the art method on the UB Kinface dataset and how this method differs from the proposed DLML method.

Chapter 4

Method: Deep Linear Metric Learning-(DLML)

Previous methods [Kohli et al. 2017], [Xia 2012], [Yan et al. 2014], [Yan et al. 2015], [Lu et al. 2014], [Xia et al. 2011], [Shao et al. 2011], [Kohli et al. 2012], have trained and evaluated their solutions on child-young parents pairs and child-old parents pairs separately. In this research the contrary is done using the proposed DLML method, making the cross-validation with the whole dataset. This approach is inspired by the fact that ConvNets are bioinspired by the human brain [LeCun et al. 2010] that is the responsible for identify kinship relations, thus they can execute intuitive tasks like kinship verification without the need to be informed what is the age difference of two people.

It is also important to highlight that the phases of the proposed method are not connected, each phase generates an output, that is later used as the input of the next phase. This decision was made in order to try to maximize flexibility during research.

4.1 Hardware and Environment

All the experiments were performed on a laptop with the following configuration:

- Processor: Intel(R) Core(TM) i7-4720HQ CPU @ 2.60GHz
- Memory: 16Gb of memory
- GPU: GTX970M

All the code developed and from third-parties was developed using Python with environment parameters to allow to run commands via shell script.

The Tensorflow version used was compiled to use multiple sets of instructions that are specific to the GPU, and it improves the performance of the GPU calculations using CUDA.

Measuring by the time necessary to perform facial alignment on the LFW images with the MTCNN architecture, the custom version provided a performance six times better than the general GPU version of Tensorflow, and 21 times better than the CPU version on the same hardware as presented on Table 4.1.

Table 4.1: Tensorflow time to process LFW dataset with MTCNN

Version	LFW time (seconds)
CPU	441
GPU	126
GPU for GTX-970M	21

TensorFlow is an open source software library for machine learning that uses data-flow graphics. Nodes represent math operations and the graph edges represent the multidimensional data arrays (tensors) that flow among them. This is a flexible architecture that allows deployment of computation to one or more CPUs or GPUs in desptops, servers, or mobile devices without the need of rewriting code. TensorFlow also includes TensorBoard, a data visualization toolkit that is used to generated graphs on this research [Community 2018].

Tensorflow 12.0 was compiled specifically for the GPU GTX-970M using CUDA 10, Python 3.7 and GCC-7.

4.2 Datasets

In this section, the datasets used in the research are explored and some of the necessary operations. All data used will be formed by unconstrained images (on the wild).

4.2.1 Training Datasets for Deep Learning

The images of all the datasets do not have a standard size, to convert and align the face of all images to the necessary 160x160px size to use on the FaceNet implementation used on this research [Sandberg 2018], the MTCNN implementation provided by [Sandberg 2018] is used. This model uses the weights of the original MTCNN [Zhang et al. 2016] on a TensorFlow implementation.

4.2.2 VGGFace2

The VGGFace2 is a large-scale face dataset with large age variations, composed of 3.31 million images of 9131 subjects. It has an average of 362.6 images for each subject [Cao et al. 2017]. Only the training portion of VGGFace2 that has 8631 classes and approximately 3.14 million images was used to train the FaceNet implementation [Sandberg 2018]. On Table 5.1 it is displayed how many images compose the training portion of the VGGFace2 dataset. The features of VGGFace2 are what initially inspired our use of young and old parents images without any special treatment to large age differences. It is assumed that the final model would be robust to large age variations because the VGGFace2 has a high variation on this aspect. 59.3% of the images of VGGFace are from male subjects as presented on Figure 4.1.

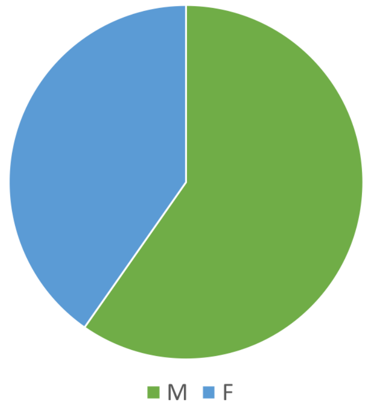


Figure 4.1: Gender balance on VGGFace2 dataset (59.3% male), (40.7% female)-source: [Cao et al. 2017]

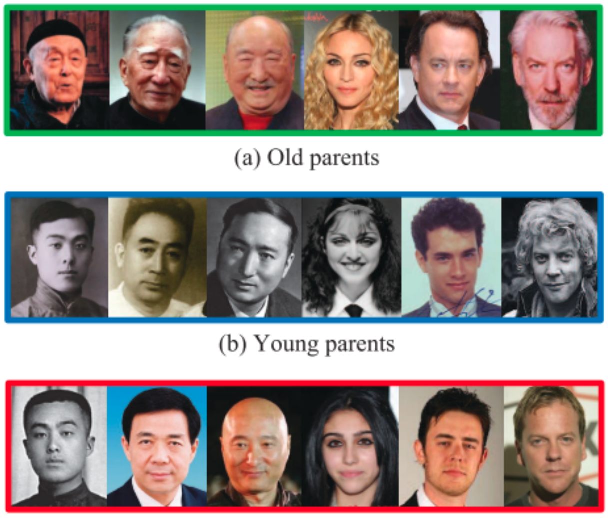
Because the data limitation of kinship datasets and the necessity of data for deep learning methods, it is necessary to use a dataset with a different main purpose than the one of this research, VGGFace2 is used to train and validate FaceNet to extract facial features. FaceNet and VGGFace2 will allow us to leverage the superiority of deep learning models stated by [Georgopoulos et al. 2018] on the kinship verification task.

4.2.3 Labeled Faces on the Wild-LFW

Labeled Faces on The Wild(LFW) was also used additionally as a test dataset for FaceNet on training. LFW has 1680 classes with two or more distinct photos [Huang et al. 2017]; these classes are used to test FaceNet performance on intervals of five epochs of training. Tests are made with this dataset because it is one of the main benchmarks for facial recognition tasks [Learned-Miller et al. 2016]. The results obtained on this tests are not used to adjust the weights of the network, only to assess the performance of the trained model without bias.

4.2.4 The UB Kinface dataset

UB KinFace dataset is used to perform cross-validation for kinship verification. The dataset is made of 200 group of images composed by old parents, young parents, and children(total of 600 images). Most of the pictures of young parents are in grayscale because of the technology available at the time of the photos; there are also other examples of isolated grayscale images. In Figure 4.2 examples of pictures from the UB Kinface dataset are exhibited.



(a) Old parents

(b) Young parents

(c) Childern

Figure 4.2: The UB Kinface dataset [Shao et al. 2011]

The types of kinship relations are not evaluated separately because nearly 80% of the relations are father-son relations as presented on the statics of the dataset at Figure 4.3.

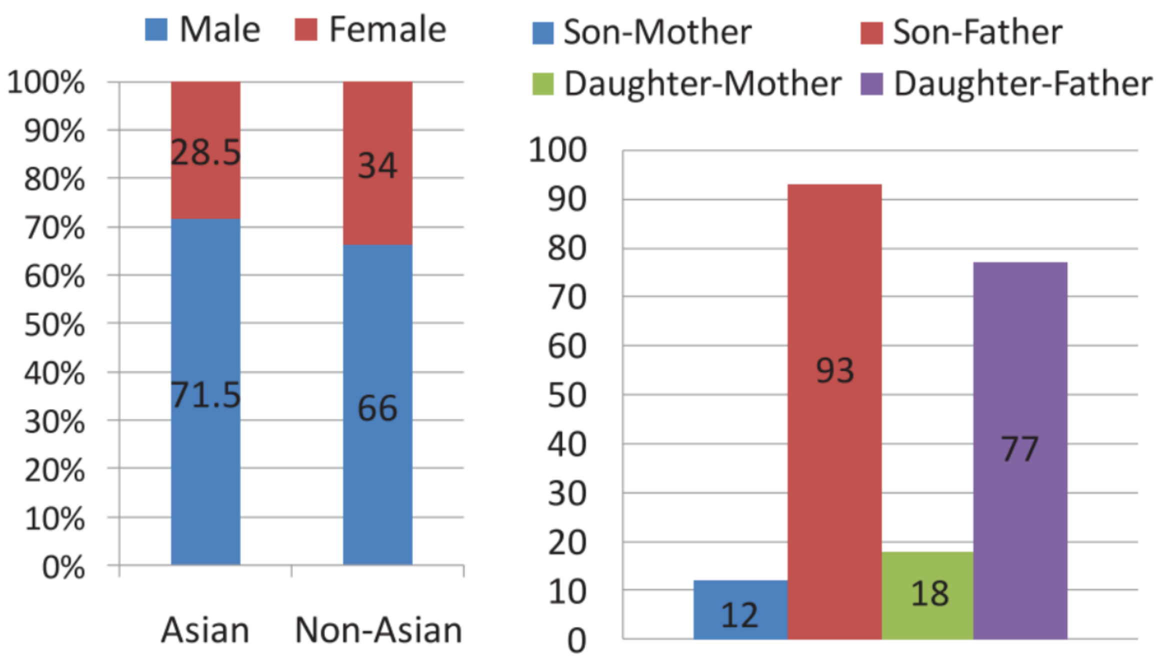


Figure 4.3: Statics of the UB Kinface dataset- source: [Shao et al. 2011]

4.3 Multi-Task Cascaded Convolutional Network (MTCNN)

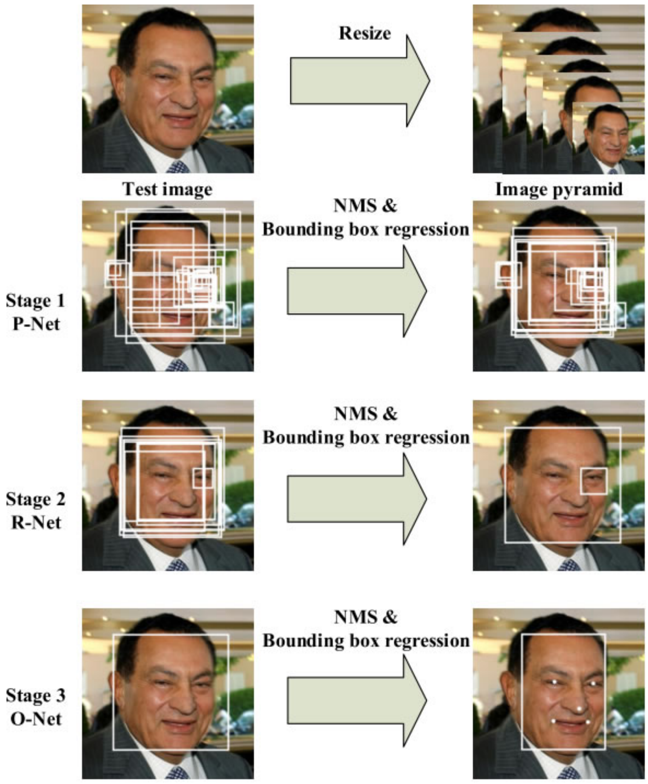


Figure 4.4: The complete three phase process of the MTCNN - source: [Zhang et al. 2016]

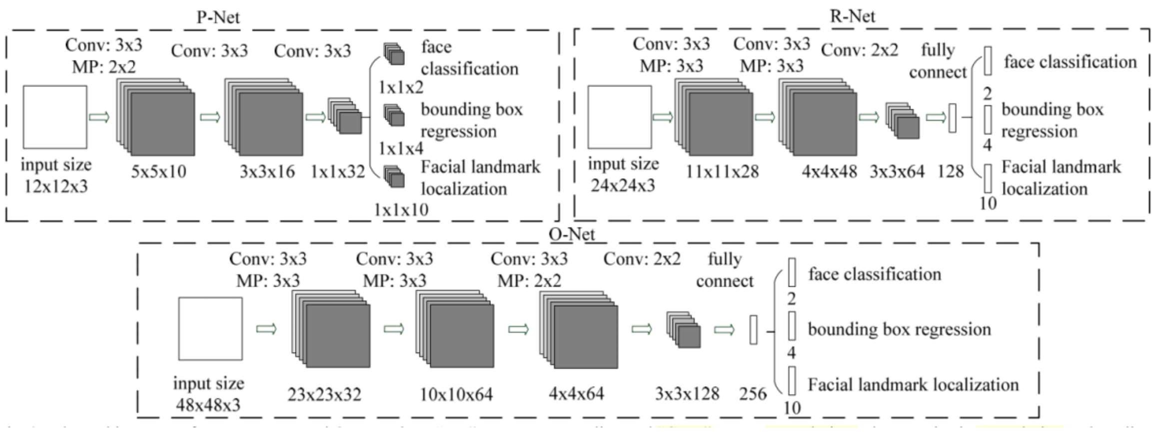


Figure 4.5: Detailed version of MTCNN architecture - source: [Zhang et al. 2016]

An MTCNN implementation [Sandberg 2018] is used to perform facial alignment because it provides good performance on hard examples like various poses, illuminations, and occlusions [Zhang et al. 2016]. The MTCNN architecture first resizes the image to different scales to build an image pyramid which will be the input of a three-phase cascade framework [Zhang et al. 2016]. The complete framework can be seen on Figure 4.4 and a detailed version of the MTCNN architecture can be seen on Figure 4.5. The MTCNN facial alignment process is described as follow.

- **Proposal Network (P-Net)** On the first stage, a fully convolutional neural network find the candidate facial windows and the bounding box regression vectors. These candidates are found estimating the borders of the face. After that, a non-maximum suppression (NMS) is applied to merge highly overlapped candidates [Zhang et al. 2016].
- **Refine Network (R-Net)** On the second stage all candidates are processed by another network which discards a significant number of false candidates, executes calibration with bounding box regression, and performs NMS [Zhang et al. 2016].
- **Identify Facial Landmark** It is similar to the second network, but in this case, the goal is to identify face regions with more supervision, providing five facial landmarks as output [Zhang et al. 2016]

The goal of the research is not performing facial alignment, so, a pre-trained model [Sandberg 2018] that uses the weights provided by the authors of the MTCNN paper [Zhang et al. 2016] is used.

The post-MTCNN images will be a stretched version of the face with the size 160x160. The reason to use the stretched face is is that the necessary features for FaceNet are kept after transformation [Schroff et al. 2015], and this allows FaceNet to have the standard input size of 160x160.The The "Original color images" are composed by these images.

These aligned face images of the UB Kinface dataset will also be used to create a new dataset that consists of all the images converted to grayscale(grayscale images. This dataset has the purpose of analyzing the impacts of different channel patterns on the results.

4.4 Converting UB Kinface to grayscale

During tests with the original images from UB Kinface, the results showed that the variance on color channel patterns present on the UB Kinface dataset (colorful and grayscale images) increased the distance between faces and decreased the performance of the linear model. To overcome this increase in distance because of color patterns a grayscale version of the UB Kinface dataset was created.

To create the grayscale version of images that are used for experiments, the algorith presented on List 6.10 is executed.

List 4.1: Algorithm that generates grayscale images

```
function create_and_save_grayscale_images(source_image_path,
    output_image_path) {
    img = load(source_image_path)
    img_gray = to_grayscale(img)
    save_image(output_image_path)
}
```

```

function process_dataset(arguments) {
    folder_path_list = load_all_folders(args.root_dir_dataset)
    output_path = arguments.output_dir_dataset
    for (folder_path in folder_path_list) {
        image_path_list = load_all_files(folder_path)
        folder_output_path = output_path + folder_path
        for (image_path in image_path_list) {
            source_image_path = folder_path + image_path
            output_image_path = folder_output_path + image_path
            create_and_save_grayscale_images(source_image_path,
                output_image_path)
        }
    }
}
```

4.5 FaceNet

The FaceNet architecture used in this research(Table 4.2) has shown one of the best performances on some of the most relevant facial recognition benchmarks like LFW and Youtube Faces Database [Schroff et al. 2015]. Another key factor that inspired the use of FaceNet is the fact that the network generates an array of facial embeddings, assuming the principle that this array can be applied to other purposes, in this research it was used to perform kinship verification.

4.5.1 Inception-ResNet-v1 modules

On the FaceNet implementation used on this research, the Inception-ResNet-v1 is used, this version reduces the computational cost and offers better performance. This Inception version also offers better accuracy and better convergence on training [Szegedy et al. 2016].

The main difference between the classical Inception and the ResNet version is the use of residual connections, that consists in using the output of the previous layer as a direct input to the next layer, with convolutions being performed on paralel [Szegedy et al. 2016]. Residual connections can be observed on Figure 4.6, Figure 4.7, and Figure 4.8.

The Inception-ResNet-v1 is composed of 3 types of inception modules(layers). The first one is the Inception-A that is presented on Figure 4.6, it has a grid of 35x35.

The second one is the Inception-B that is presented on Figure 4.7, it has a grid of 17x17.

The third one is the Inception-C that is presented on Figure 4.8, it has a grid of 8x8.

To extract features on FaceNet, convolutional, pooling, and inception layers are used as shown at Table 4.2. The convolutional and pooling are done on the first stage; the inception

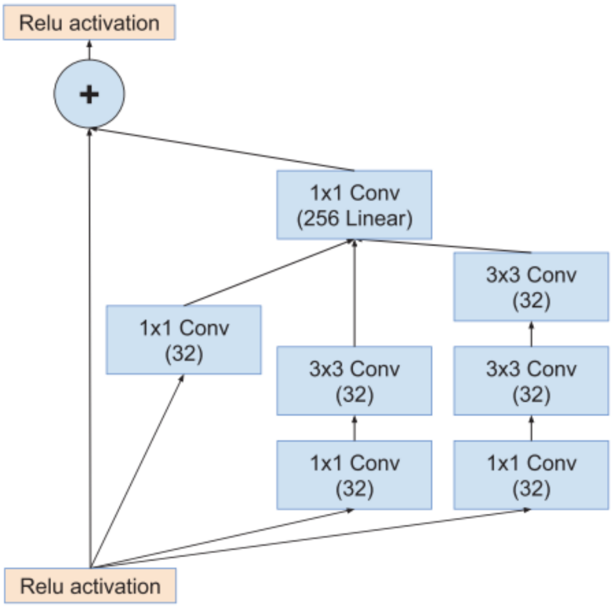


Figure 4.6: Inception-A layer for Inception-ResNet-v1 - source: [Szegedy et al. 2016]

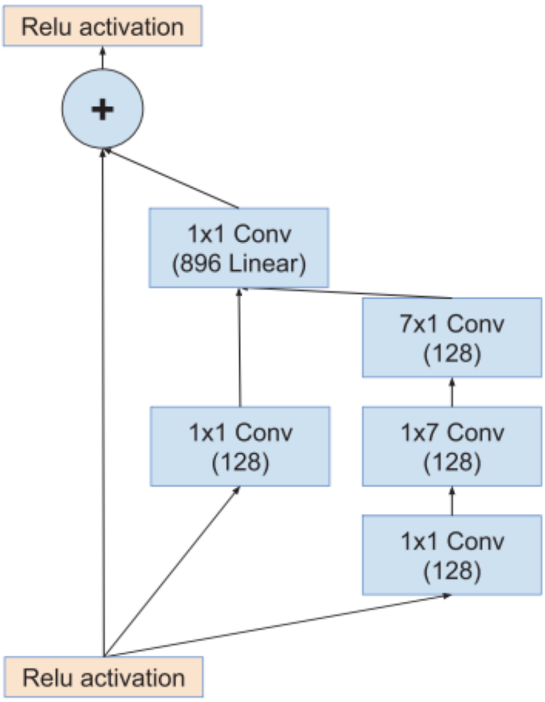


Figure 4.7: Inception-B layer for Inception-ResNet-v1 - source: [Szegedy et al. 2016]

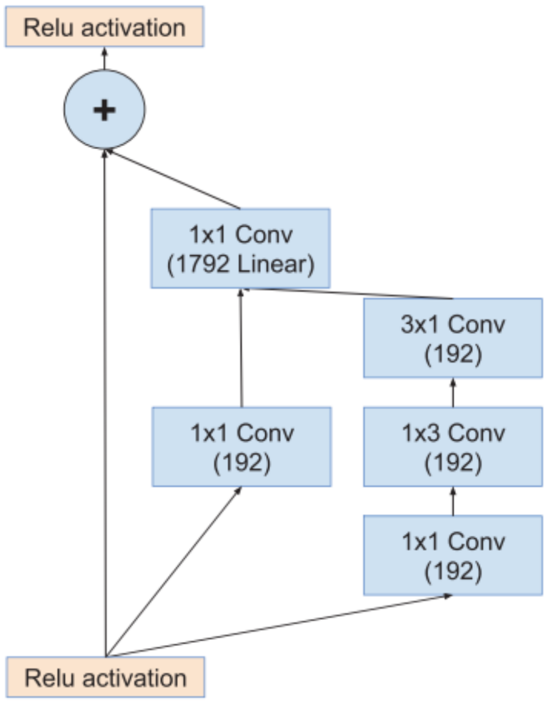


Figure 4.8: Inception-C layer for Inception-ResNet-v1 - source: [Szegedy et al. 2016]

layer is responsible for extracting mid-level features.

The Reduction-A(layer 13) and Reduction-B(layer 23) layers on Table 4.2 are basically elaborated pooling layers.

FaceNet creates an array of 128 dimensions of the face; those dimensions are used on training to create an abstract representation of the face that it is called anchor. The anchor will have a maximum distance of all representations of that face. It is possible to see the FaceNet method in a simple perspective on Figure 4.9.

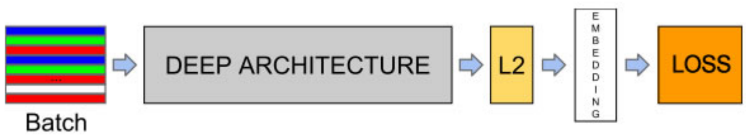


Figure 4.9: FaceNet original model structure [Schroff et al. 2015]

On training, the original paper [Schroff et al. 2015] used a triplet loss function. The training process increases the distance of negative samples and approximates the positive samples as shown in Figure 4.10 [Schroff et al. 2015].

Triplet loss is computationally more costly than training as a softmax classifier using cross-entropy loss and training as a classifier can still offer good results [Parkhi et al. 2015]. On this research, Facenet was trained as a softmax classifier with cross-entropy loss.

On the original FaceNet paper the architecture used was a non-ResNet version

Table 4.2: The FaceNet architecture used on this research

stage	layer type
1 to 3	3 x Convolution
4	Max pooling
5 to 7	3 x convolution
8 to 12	5x Inception-A
13	Reduction-A
13 to 22	10x Inception-B
23	Reduction-B
24 to 28	5x Inception-C
29	Average pooling
30	Flattening layer
31	Fully connected

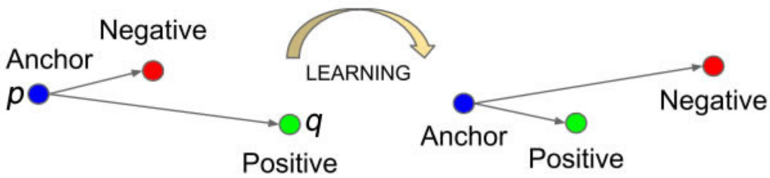


Figure 4.10: Anchors on the training process [Schroff et al. 2015]

of the inception architecture [Schroff et al. 2015]. In this research, the Inception-Res1Net-v1 architecture is used because it provides better performance and convergence [Szegedy et al. 2016]. The FaceNet implementation used is based on the NN3 architecture of the original paper [Schroff et al. 2015]. This network has input size of 160x160.

To perform testing on the LFW dataset on every epoch of training, the "*Pair Matching*" protocol with "*Unrestricted, with labeled outside data*" provided by [Huang and Learned-miller 2014] is used. The 1680 classes with more than two images are used to form pairs of images without overlapping. These pairs will test the distance between the two embeddings created by the network. A class with four images, for instance, will have two pairs of images to evaluate, [0,1] and [2,3]. This distance is calculated using the euclidean distance between the two embeddings(L2 norm) as described on Eq. 4.1, with p_i as one of the embeddings and q_i as the other.

$$[htpb]L2 = \sqrt{\sum_{i=0}^{127}(p_i - q_i)^2} \tag{4.1}$$

The test results on LFW during training are not used to adjust the network weights, only to assess the performance of FaceNet.

4.6 Linear metric learning for kinship verification

The last stage tackles the fact that facial features of similar people lie in a close neighborhood, but this does not necessarily mean that these two people are kin, and the contrary is also true. Enters the last phase of our method with the metric learning approach that tries to learn what are the right feature differences to detect kin and a non-kin people.

The extracted features of two images are subtracted forming positive difference pairs like [[1,201], [2,402], ...], and negatives such as [[1,225], [2, 561]]. Considering 1 to 200 as children, 201 to 400 as young parents, and 401 to 600 as old parents.

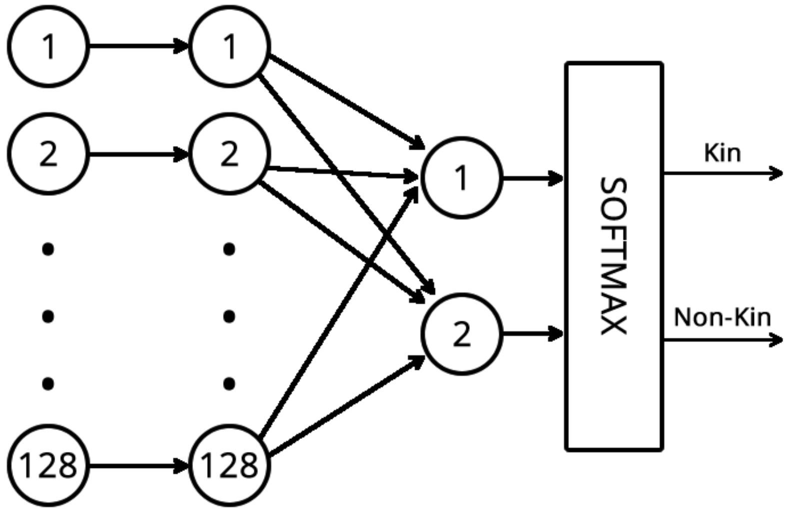


Figure 4.11: The linear model that receives the non-negative difference array

List 4.2: Function that generates the model

```
function create_network(input_array, size_input = 128, n_classes = 2,
    keep_prob = 0.4) {
    layer_1 = dense_layer(input_array, size_input, activation=
        leaky_relu)
    dropout_layer = dropout(layer_1, keep_prob)
    layer_2 = dense_layer(dropout_layer, n_classes, activation=
        leaky_relu)
    prediction = softmax(layer_2)
    return prediction
}
```

The non-negative result of the subtraction is fed into the linear model of Figure 4.11 that it will perform the kinship verification. The same model is also presented at List 6.5. This model has 128 inputs (same size as the embeddings provided by FaceNet). The first layer has the size of 128x1 with bias unities, and it uses the Leaky Rectified Linear Unit(Leaky ReLu) activation function [Xu et al. 2015].

Next, a fully connected layer with only two outputs finalizes the model, also using the Leaky ReLU activation function. Finally, a softmax function is used to perform the boolean prediction of kin or non-kin.

On training, dropout [Goodfellow et al. 2016] is applied after the first layer, the cross-entropy loss function showed on Eq. 4.2 and List 6.6 is used being y_i the predicted value provided by the model, and y_i^l as the expected value. The classical standard backpropagation algorithm with gradient descent [LeCun et al. 1989] performs optimization of the network during training as shown by the function List 6.7 that it creates the optimizer.

$$-\sum_i y_i^l \cdot log(y_i) \tag{4.2}$$

List 4.3: Cost function for the linear model

```
function create_cost_function(model, expected_value) {
    cost_function = sum(expected_value * log(model)
    return cost_function
}
```

List 4.4: Optimizer for the linear model

```
function create_optimizer(learning_rate, cost_function) {
    optimizer = GradientDescentOptimizer(learning_rate, cost_function)
    return optimizer
}
```

4.7 Final considerations

This chapter presents the hardware used and all the datasets used on this research(VGGFace2, LFW, and UB Kinface), the number of images on these datasets is explored, some of specific the characteristics of these datasets are presented, and the available statistics of each dataset are exhibited.

On this chapter it is also presented the proposed DLML method is explained in detail, talking about the specifics of each architecture (MTCNN, FaceNet, linear metric learning model). The fact that MTCNN is not trained as part of this research is explained. The training of the FaceNet architecture is explained, and a comparison between the softmax method used by this research and the triplet loss from original paper is made. The linear metric learning model is explained in detail, and the main algorithms are presented.

Chapter 5

Experiments and results

This section will explore the tasks and experiments made in this research, show and discuss the results of these experiments.

5.1 Face Alignment

Even though face alignment is not a part of the main purpose of this research, it is a necessary step to perform feature extraction with FaceNet and kinship verification with the linear model. MTCNN was the architecture choosed because deep ConvNets architectures, to the best of our knowledge, are the state of the art solution to deal with unconstrained images [Goodfellow et al. 2016], MTCNN has also consistently outperformed the state-of-the-art methods across several challenging benchmarks [Zhang et al. 2016]. MTCNN was also used on the original VGGFace2 article [Cao et al. 2017] to perform facial alignment.

Since face alignment is not one of the main tasks of this research, the model used was the one that it is provided with the FaceNet implementation used on this research [Sandberg 2018]; this model is implemented using Tensorflow, the original is implemented on Matlab [Zhang et al. 2016]. However, the authors from the original paper published the original code and model as open-source, and the implementation used in this research imports the weights of the original model into the new implementation.

Facial alignment is performed using MTCNN for three datasets: VGGFace2(FaceNet training), LFW(FaceNet testing), and UB Kinface (kinship verification cross-validation). To process the images of the datasets MTCNN is executed for each dataset.

That is a *–margin* option on the implemented code, the margin would add additional space between the border and the detected face of the image. The margin option was kept as 0 because about half of the images on UB Kinface have face area smaller than 160x160px, adding a margin would reduce even further the quality of half of the post-MTCNN facial images, that it already had to be reduced on 311 of 600 cases(51.83%) to stretch to 160x160px.

The post-MTCNN images of all datasets will have 160x160px, that size is necessary to use the images as input on the FaceNet architecture, the images will be a stretched version of the aligned face to fit on this dimension. On Figure 5.1 it is possible to see samples of images from UB Kinface before and after facial alignment. The MTCNN implementation used also provides information about the facial bounding boxes detected on the images; this allows calculating the size of the facial area for each image.

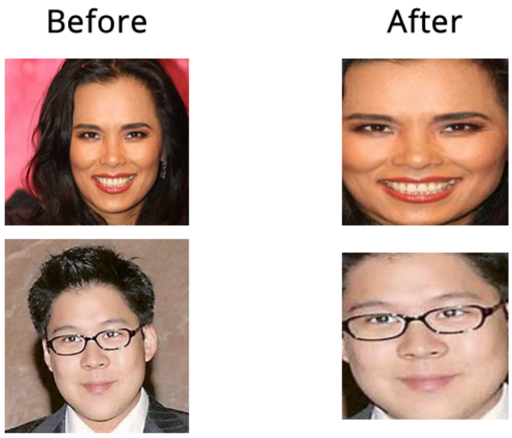


Figure 5.1: Image samples from the UB Kinface database before and after facial alignment with MTCNN

The performance of the MTCNN pre-trained model [Sandberg 2018] on the three datasets is exhibited on Table 5.1, the before column shows how many images were available before facial alignment, the after shows how many were successfully aligned, and the performance is calculated by comparing how many of the images were processed successfully.

Table 5.1: Performance of MTCNN on datasets

	Before	**After**	**Performance**
VGGFace2	3,141,890	3,138,862	99.90%
LFW	13,233	13,233	100%
UB Kinface	600	600	100%

From a total of 3,155,723 images 3,152,695(99.90%) images were successfully aligned as presented on Table 5.1. These results showed that the MTCNN architecture performed well on the unconstrained(on the wild) image data used on this research; the post-MTCNN images will allow the next necessary steps(feature extraction and kinship verification) to take place.

5.2 Feature extraction with FaceNet

This section will discuss the training, testing, validation, and use of FaceNet on this research to extract features from facial images.

Because of the specific size and border used on our research to avoid decreasing in image quality, and the large age variation present on our data for kinship verification (UB Kinface) we had to train our own model to perform feature extraction with VGGFace2, this model is tested on LFW and it is responsible for creating the 128x600 dimensions array of features of the 600 images of the UB Kinface dataset.

5.2.1 Training

FaceNet is trained with a total of 500 epochs, each epoch has 1000 batches and each batch has 40 images.

To improve performance and avoid overfitting the fixed image standardization(normalization) [Goodfellow et al. 2016] technic and dropout [Goodfellow et al. 2016] are used. Table 5.2 displays the empirical learning rate used for training with the Adam optimizer [Kingma and Ba 2015].

Table 5.2: Empirical learning rate for FaceNet training

Epoch	Learning Rate
0-99	0.1
100-299	0.05
300-399	0.005
400-499	0.0005

5.2.2 Validation

A portion of 0.01% of the VGGFace2 is used for validation on training to calculate the loss and adjust the weights using the Adam optimizer [Kingma and Ba 2015]. The total sum of the cross-entropy loss can be seen in Figure 5.3. The cross entropy loss value of every batch can be observed in Figure 5.2.

5.2.3 Testing

The accuracy of testing on LFW exhibited at Figure 5.4 is calculated every five epochs using the euclidean distance. After completing training, tests are run again on LFW using the euclidean distance; the accuracy was 98.83%.

After trained with a softmax classifier at the last layer [Parkhi et al. 2015], FaceNet was able to successfully detect and export the features for the 600 images of the UB Kinface

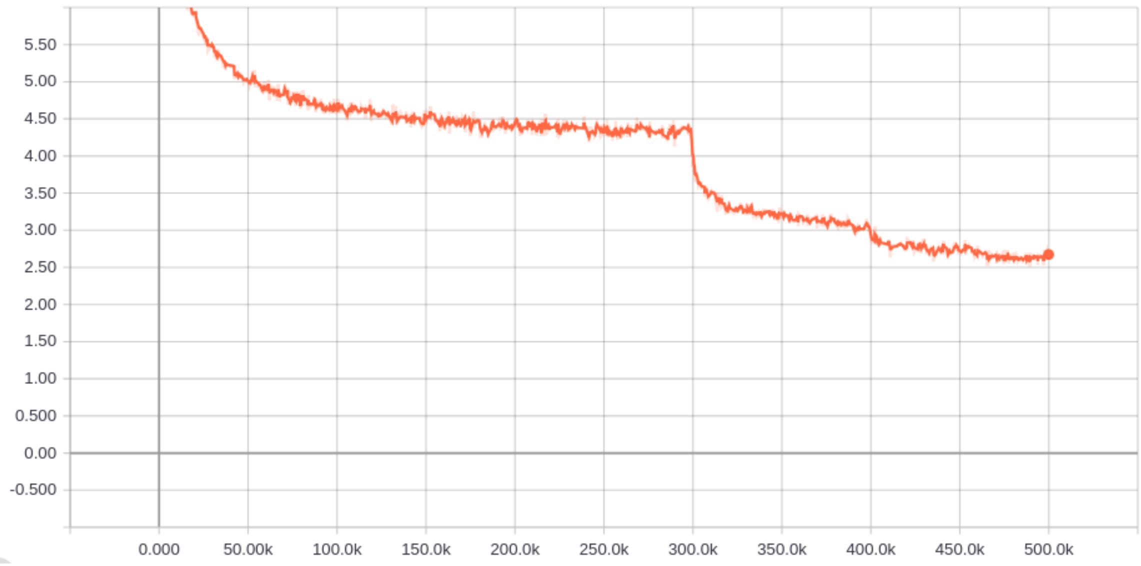


Figure 5.2: Cross entropy on training using VGGFace2

Figure 5.3: Total loss on training using VGGFace2

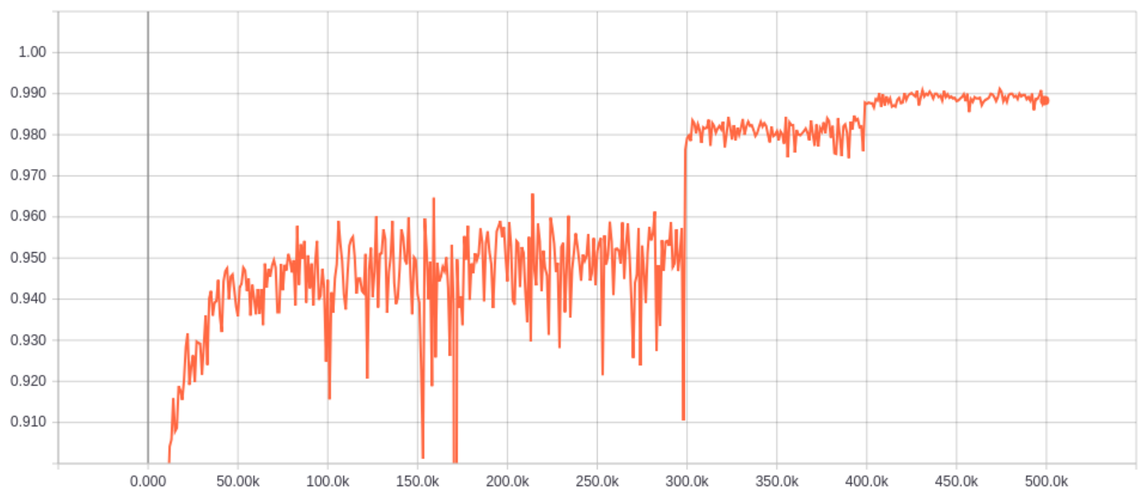


Figure 5.4: Testing accuracy on LFW every five epochs

Table 5.3: Sample of the extracted features from the facial images on the UB Kinface using FaceNet

	Feature 0	Feature 1	...	Feature 127
Image 0 (Child)	-0.027277473360300064	-0.08217132836580276	...	0.06560494750738144
...	...	...	...	...
Image 200 (Young-parent)	0.10031850636005402	-0.0060198549181222916	...	0.08803482353687286
...	...	...	...	...
Image 230 (Young-parent)	-0.0584646500647068	-0.12933146953582764	...	0.08840493857860565
...	...	...	...	...
Image 400 (Old-parent)	0.15667474269866943	-0.13348889350891113	...	-0.04572216048836708
...	...	...	...	...
Image 493 (Old-parent)	-0.016640465706586838	0.03600674122571945	...	-0.10782409459352493
...	...	...	...	...
Image 600 (Old-parent)	-0.060755062848329544	-0.03715011477470398	...	0.0024316716007888317

dataset. The values will range from -1 to 1 because of normalization, and this values will be used to create the non-negative array that it will serve as input for the linear metric learning model. On Table 5.3 it is possible to see the structure of the extracted features and a sample of the values.

When presented with a grayscale image, FaceNet will basically represent the grayscale image in three color channels, keeping the grayscale aspect of the data. Even though FaceNet was trained and tested with colorful images from the VGGFace2 and LFW datasets respectively, the results on the kinship-verification cross-validation showed that FaceNet was able to extract expressive features from grayscale images.

5.3 Cross-validation on the kinship verification linear metric learning model

Because of the size of UB Kinface dataset, it is recommended to use cross-validation methods to perform tests [Georgopoulos et al. 2018], since if tested with only one model, the results could easily be biased because of the small amount of data. Because of the cross-validation approach, there are also multiple models instead of just one.

The dataset used for cross-validation is composed of 800 image features pairs as described on Table 5.4, the dataset is mounted in a balanced way that follows the structure: [true child-old parent, false child-old parent, false child-young parent, true child-young parent, true child-old parent, ...].

The negative child-young parent's pairs are composed of non-kin pairs between child and other young parent individuals, the negative child-old parent's pairs are made of non-kin pairs of child and old parent individuals. This pattern will repeat throughout the 800 available samples to assure that the tests are well balanced between the four types of examples, that way on each cross-validation cycle, there will always be a balanced number of type examples on training and testing.

Table 5.4: Number of examples used for cross-validation

Type of Relations	Positive	Negative
Child-young parent	200	200
Child-old parent	200	200

Table 5.5: Non-negative distance array that is used as input for the linear model

Pair	Type	Distance 0	Distance 1	...	Distance 127
0-400	True Child-Old Parent	0,18395221605897	0,066887825727463	...	0,111327107995749
0-493	False Child-Old Parent	0,010637007653713	0,102607809007168	...	0,173429042100906
0-230	False Child-Old Parent	0,031187176704407	0,062730401754379	...	0,022799991071224
0-200	True Child-Young Parent	0,127595979720354	0,060581212863326	...	0,022429876029492
1-401	True Child-Old Parent	0,102011129260063	0,246223147958517	...	0,049301842227578
...		...	...	...	...

A sample of the non-negative array that measures the distance between image features of the UB Kinface is presented on Table 5.5; The array will repeat the presented structure on every 4 items until it reaches the combination of number 800, it is also possible to observe that the child image is used as anchor to generate the 4 types of relations for each iteration. This structure is what guarantees that the data for cross-validation-cycles have a balanced number of types of examples on the training and testing parts.

The cross-validation data is created by iterating through the array of 800 distances embeddings in an orderly manner. For the leave-one-out protocol, for example, 80 images are used for test per cycle: cycle 1: 0-79, cycle 2: 80-159, ... ; The rest of the data is always used for training. Because of the balanced aspect of the dataset, the cross-validation cycle will not be biased to a specific type of relation. For the five-fold-cross validation, the same process is executed but with 160 images for test per cycle.

Table 5.6: Training parameters for the linear model

Learning Rate	Epochs	Batch Size	Dropout
0.07	10	10	0.4%

The 800 pairs of images of the UB Kinface are tested with the 5-fold cross-validation protocol and leave-one-out protocol with the original images and grayscale images; these tests will create a total of 15 different models (10 for leave-one-out and 5 for five-fold) for original images and 15 models for the grayscale images. The training parameters used to train the models are showed on Table 5.6.

The results for all the leave-one-out cycles can be seen on Table 5.7, and the results for all the five-fold cycles are presented on Table 5.8.

On Table 5.7, it is possible to observe that the leave-one-out protocol with grayscale images offered the best accuracy on very cycle of tests, it is also possible to see decrease in accuracy, precision and recall with the original images because of heterogeneous color patterns present on the UB Kinface dataset, this characteristic also generated a few outliers

Table 5.7: Results of all the leave-one-out cross-validation cycles with original and grayscale images

Leave-one-out cross-validation results							
		Original images			Grayscale images		
Cycle	Test pair interval	Accuracy	Precision	Recall	Accuracy	Precision	Recall
0	0-79	0.488	0.491	0.700	0.725	0.688	0.825
1	80-159	0.438	0.368	0.175	0.788	0.795	0.775
2	160-239	0.462	0.467	0.525	0.675	0.646	0.775
3	240-319	0.462	0.468	0.550	0.662	0.638	0.750
4	320-399	0.400	0.278	0.125	0.700	0.674	0.775
5	400-479	0.412	0.360	0.225	0.638	0.593	0.875
6	480-559	0.512	0.516	0.535	0,725	0.705	0.775
7	560-639	0.488	0.490	0.600	0.750	0.685	0.925
8	640-719	0.438	0.429	0.375	0.725	0.680	0.850
9	720-799	0.563	0.568	0.525	0.762	0.698	0.925

Table 5.8: Results of all the five-fold cross-validation cycles with the original and grayscale images

Five-fold cross-validation results							
		Original images			Grayscale images		
Cycle	Test Pair Interval	Accuracy	Precision	Recall	Accuracy	Precision	Recall
0	0-159	0.412	0.436	0.600	0.681	0.704	0.625
1	160-319	0.419	0.240	0.750	0.688	0.636	0.875
2	320-479	0.400	0.389	0.350	0.681	0.649	0.788
3	480-639	0.431	0.441	0.513	0.625	0.601	0.750
4	640-799	0.419	0.440	0.600	0.694	0.663	0.788

on the original images with precision and recall (0.278 for precision and 0.125 for recall). Most of the cycles had similar results, on the grayscale images, for instance, the maximum accuracy difference between two cycles is 0.15 (15.00%), the maximum precision difference is 0.157 (15.70%), and the maximum recall distance is 0.175 (15.50

The results presented for accuracy with grayscale images at Table 5.7 showed that the proposed method, on every cycle of the cross-validation process, has better accuracy then the human baseline, consistently performing kinship verification on the UB Kinface dataset; The precision values show that the DLML proposed method offers better than 60% performance on 90% of the cycles to detect positive kinship pairs when considering all positive pairs on the data, that means that on most cycles the model is correct on 60% of times. The recall values showed that on every cycle the model identifies above 75% of positive kinship relations.

On Table 5.8 it is possible to observe that on the five-fold cross-validation tests the model performs better with the homogeneous grayscale color dataset created, accuracy and precision are always above 60.00% , that means that on more than 60.00% of time the answer is right, and more than 60% of the positive examples classified the model are correct. Considering the recall, it is also possible to affirm that on 80.00% of cycles the model correctly classifies a kinship relation on more than 70.00% of cases.

The overall experiment results are exhibited on Table 5.9 and Table 5.10, the exhibited values are the average of all the values obtained during the cross-validation cycles.

Table 5.9: Five-fold cross-validation overall results

5-fold			
	Accuracy	Precision	Recall
Original color images	41.63%	38.93%	42.75%
Grayscale images	67.38%	65.06%	76.50%

Observing the five-fold cross-validation grayscale images results presented by Table 5.9, it is possible to observe the models are right on average on 67.38% of occasions, that the models are right on average on 68.01% of cases classified as a positive kinship-relation between pairs. It is also possible based on the recall to observe that the models classified correctly 76.50% of all the positive cases.

Table 5.10: Leave-one-out, overall cross-validation results

Leave-one-out			
	Accuracy	Precision	Recall
Original color images	46.62%	44.34%	42.00%
Grayscale images	71.50%	68.01%	82.50%

Considering the leave-one-out cross-validation grayscale images results presented by Table 5.10, it is possible to observe the models are right on average on 71.50% of occasions, that on average the models are right on 68.01% of cases classified as a positive kinship-relation between pairs. It is also possible based on the recall values to observe that the models classified correctly 82.50% of all the positive cases.

The results presented on Table 5.9 and Table 5.10 confirmed that the heterogeneous color pattern of the images on the UB Kinface dataset compromised the performance of the linear metric learning model, creating additional distance between the arrays of features of the images. The grayscale images showed that with only features extracted from grayscale images it is possible to surpass the human baseline and other methods evaluated on the UB Kinface dataset.

5.4 Final considerations

On this chapter the actions performed on this research are explored in detail, the reasons for the use of each technic are presented, all the training processes are explained and the available data presented. There is also explanations and samples of how the cross-validation datasets are created and organized, and examples of how images are processed by the MTCNN implementation.

The results of all the cross-validation cycles the experiments are presented and the meaning of these results are explored in detail by making comparisons between grayscale and colorful images. These results also justify decisions taken during this research.

Chapter 6

Conclusions

Our results showed that on the UB Kinface database our Deep Linear Metric Learning method can be used to solve the kinship verification problem, even when there are large age differences, increasing the applicability of the model in a real-world environment.

The proposed method shows robustness to the mix of old and new image data present in the database. The presented method performs directly on kinship verification with large age variations, without the need for retraining relations separately on child-young parents and child-old parents as seen in other approaches.

By comparing the results between the original color images and the grayscale images on Table 5.9 and Table 5.10, it is possible to verify that even though the features are extracted with a network that it is trained with colorful facial images, the difference of the extracted features provided by FaceNet, when dealing with pair of images in color and grayscale, decreases the performance of the proposede linear model because the distance created by different color channel patterns impacts how expressive the features are to the linear model. This difference led to a worse performance on the images with the original color (colorful and grayscale mixed) than when all images are converted to grayscale.

Despite FaceNet being trained with colorful images, it provides good feature extraction for grayscale images of the UB Kinface database, since these features allowed the linear metric learning stage to achieve good performance with grayscale images.

Comparing the achieved results on Table 5.9 and Table 5.10 with the results of other methods on Table 3.1, it is possible to observe that our proposed DLML method has very similar accuracy to the fourth best method PDFL with 5-fold cross-validation, 67.30% against 67.38% of the presented method.

With the leave-one-out protocol, our method ranks as the best performance with 71.50% of accuracy. Our DLML method is also superior to the human baseline performance of 56.00%; These results showed that the Deep Linear Metric Learning approach can be used in the kinship verification with large age variations without tackling separately large age differences. Finally, by discarding the necessity of detecting and treating large age differences our method offers an enhanced all-in-one solution to the kinship verification problem.

6.1 Future Work

The proposed solution showed promising results on the dataset for kinship verification with a large age variation. Further and larger datasets will continue to become available, and for sure further testings would be necessary, especially in order to try to evaluate mother and father's different influences on facial inherited features.

Explore the results of other methods with the same or similar approach in order to better assess the performance of our method.

Explore and develop other training methods to extract different features and possibly combine layers for evaluating performance on different subset problems.

Bibliography

[Cao et al. 2017] Cao, Q., Shen, L., Xie, W., Parkhi, O. M., and Zisserman, A. (2017). VGGFace2: A dataset for recognising faces across pose and age.

[Chergui et al. 2018] Chergui, A., Ouchtati, S., and Bougourzi, F. (2018). LPQ and LDP Descriptors with ML Representation For Kinship verification. *International Workshop on Signal Processing Applied to Rotating Machinery Diagnostics*, 2(April).

[Community 2018] Community (2018). Tensorflow. Available: `https://github.com/tensorflow/tensorflow`. Accessed in: 10 of December of 2018. [Online].

[Dehghan et al. 2014] Dehghan, A., Ortiz, E. G., Villegas, R., and Shah, M. (2014). Who do i look like? Determining parent-offspring resemblance via gated autoencoders. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 1757–1764.

[Fukushima 1980] Fukushima, K. (1980). Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, 36(4):193–202.

[Georgopoulos et al. 2018] Georgopoulos, M., Panagakis, Y., and Pantic, M. (2018). Modelling of Facial Aging and Kinship: A Survey. (2009):1–25.

[Goodfellow et al. 2016] Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press.

[Guo et al. 2016] Guo, Y., Zhang, L., Hu, Y., He, X., and Gao, J. (2016). MS-Celeb-1M: A Dataset and Benchmark for Large-Scale Face Recognition. *European Conference on Computer Vision*, (February):87–102.

[Gutstein et al. 2008] Gutstein, S., Fuentes, O., and Freudenthal, E. (2008). Knowledge Transfer in Deep Convolutional Neural Nets. *International Journal on Artificial Intelligence Tools*, 17(03):555.

[Huang and Learned-miller 2014] Huang, G. B. and Learned-miller, E. (2014). Labeled faces in the wild : Updates and new reporting procedures. *University of Massachusetts Amherst Technical Report*, pages 14–003.

[Huang et al. 2017] Huang, Z., Yu, Y., Gu, J., and Liu, H. (2017). An Efficient Method for Traffic Sign Recognition Based on Extreme Learning Machine. *IEEE Transactions on Cybernetics*, 47(4):920–933.

[Kingma and Ba 2015] Kingma, D. P. and Ba, J. L. (2015). Adam: a Method for Stochastic Optimization. *International Conference on Learning Representations 2015*, pages 1–15.

[Kohli et al. 2012] Kohli, N., Singh, R., and Vatsa, M. (2012). Self-similarity representation of Weber faces for kinship classification. *2012 IEEE 5th International Conference on Biometrics: Theory, Applications and Systems, BTAS 2012*, pages 245–250.

[Kohli et al. 2017] Kohli, N., Vatsa, M., Singh, R., Noore, A., and Majumdar, A. (2017). Hierarchical representation learning for kinship verification. *IEEE Transactions on Image Processing*, 26(1):289–302.

[Learned-Miller et al. 2016] Learned-Miller, E., Huang, G. B., RoyChowdhury, A., Li, H., and Hua, G. (2016). Labeled faces in the wild: A survey. *Advances in Face Detection and Facial Image Analysis*, pages 189–248.

[LeCun et al. 1989] LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., and Jackel, L. D. (1989). Backpropagation Applied to Handwritten Zip Code Recognition. *Neural Computation*, 1(4):541–551.

[LeCun et al. 2010] LeCun, Y., Kavukcuoglu, K., and Farabet, C. (2010). Convolutional networks and applications in vision. *ISCAS 2010 - 2010 IEEE International Symposium on Circuits and Systems: Nano-Bio Circuit Fabrics and Systems*, pages 253–256.

[Lu et al. 2014] Lu, J., Zhou, X., Tan, Y. P., Shang, Y., and Zhou, J. (2014). Neighborhood repulsed metric learning for kinship verification. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 36(2):331–345.

[McCulloch and Pitts 1943] McCulloch, W. S. and Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, 5(4):115–133.

[Pan and Yang 2010] Pan, S. J. and Yang, Q. (2010). A survey on transfer learning. *IEEE Transactions on Knowledge and Data Engineering*, 22(10):1345–1359.

[Parkhi et al. 2015] Parkhi, O. M., Vedaldi, A., and Zisserman, A. (2015). Deep Face Recognition. *Procedings of the British Machine Vision Conference 2015*, (Section 3):41.1–41.12.

[Ranjan et al. 2016] Ranjan, R., Sankaranarayanan, S., Castillo, C. D., and Chellappa, R. (2016). An All-In-One Convolutional Neural Network for Face Analysis.

[Rosenblatt 1958] Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408.

[Sandberg 2018] Sandberg, D. (2018). Face recognition using tensorflow.˙ Available: `https://github.com/davidsandberg/facenet`. Accessed in: 30 of July of 2018. [Online].

[Schroff et al. 2015] Schroff, F., Kalenichenko, D., and Philbin, J. (2015). FaceNet: A unified embedding for face recognition and clustering. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 07-12-June:815–823.

[Shao et al. 2011] Shao, M., Xia, S., and Fu, Y. (2011). Genealogical face recognition based on UB KinFace database. *IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, pages 60–65.

[Szegedy et al. 2016] Szegedy, C., Ioffe, S., Vanhoucke, V., and Alemi, A. (2016). Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning. pages 4278–4284.

[Szegedy et al. 2015] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. (2015). Going deeper with convolutions. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 07-12-June:1–9.

[Widrow and Hoff 1960] Widrow, B. and Hoff, M. M. E. (1960). Adaptive Switching Circuits. *1960 IRE WESCON Convention Record*, (4):96 – 104.

[Xia 2012] Xia, S. (2012). Toward Kinship Verification Using Visual Attributes. *International Conference on Pattern Recognition*, 1(Icpr):549–552.

[Xia et al. 2011] Xia, S., Shao, M., and Fu, Y. (2011). Kinship verification through transfer learning. *IJCAI International Joint Conference on Artificial Intelligence*, pages 2539–2544.

[Xu et al. 2015] Xu, B., Wang, N., Chen, T., and Li, M. (2015). Empirical Evaluation of Rectified Activations in Convolutional Network.

[Yan et al. 2014] Yan, H., Lu, J., Deng, W., and Zhou, X. (2014). Discriminative multimetric learning for kinship verification. *IEEE Transactions on Information Forensics and Security*, 9(7):1169–1178.

[Yan et al. 2015] Yan, H., Lu, J., and Zhou, X. (2015). Prototype-Based Discriminative Feature Learning for Kinship Verification. *IEEE Transactions on Cybernetics*, 45(11):2535–2545.

[Zhang et al. 2016] Zhang, K., Zhang, Z., Li, Z., Member, S., Qiao, Y., and Member, S. (2016). Joint Face Detection and Alignment using Multi - task Cascaded Convolutional Networks. *Spl*, (1):1–5.

Appendix

List 6.1: Complete linear model code

```
from __future__ import print_function

import datetime
import os
import sys
import argparse
import face_distance_calc
import dataset
import tensorflow as tf
from export_embeddings import save_file_npy_csv
import numpy as np

os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'

TXT_TRAIN_SUFFIX = ' train'
TXT_TEST_SUFFIX = ' test'
TXT_MODEL = 'Models'
TXT_RESULTS_ARRAY_SUMM_FILE_NAME = 'results_array_summ.npy'
TXT_HP_ARRAY_FILENAME = 'hp_array.npy'

tf.summary.FileWriterCache.clear()


def parse_arguments(argv):
    parser = argparse.ArgumentParser()
    parser.add_argument('data_dir', type=str,
                        help='Enter the directory with the embeddings')
    parser.add_argument('dataset_type', type=int,
                        help='Enter the type of dataset to process \n 0:
                            KinFaceV2\n 1: IMDB or CACD2000')
    parser.add_argument('--distance_metric', type=int,
                        help='Enter how you wish to calculate the
                            distance' + \
                              '\n 0: Euclidean Distance\n 1: Cosine
                                Similarity',
                        default=0)
    parser.add_argument('--distances_name_kinfacev2', type=str,
```

```
                            help='Enter the name of the archive with the distances of the KinfaceV2 dataset',
                            default=face_distance_calc.DISTANCE_LIST_NAME_KINFACEV2)
    parser.add_argument('--n_folds', type=int,
                            help='Enter the number of folds you would like to test',
                            default=5)
    parser.add_argument('--learning_rate', type=float,
                            help='Enter the learning rate value that you would like to use during trainning, default is 0.03',
                            default=0.03)
    parser.add_argument('--n_epoch', type=int,
                            help='Enter the number of epochs you would like to use on training',
                            default=15)
    parser.add_argument('--batch_size', type=int,
                            help='Enter the batch size you would like to use on training',
                            default=20)
    parser.add_argument('--size_input', type=int,
                            help='Enter the input size of the data',
                            default=128)
    parser.add_argument('--n_classes', type=int,
                            help='Enter number of classes on data',
                            default=2)
    parser.add_argument('--display_step', type=int,
                            help='Enter number steps that it will take to show the loss',
                            default=2)
    parser.add_argument('--keep_prob', type=float,
                            help='Enter the dropout value that you would like to use during trainning, default is 0.3',
                            default=0.3)
    parser.add_argument('--verbose', type=str2bool,
                            nargs='?',
                            help='False to not show model details, True to print details ',
                            default=False)
    parser.add_argument('--store', type=str2bool,
                            nargs='?',
                            help='False store model, True to print details',
                            default=False)
    parser.add_argument('--exploration_mode', type=str2bool,
                            nargs='?',
                            help='Explore the best hyper parameters',
                            default=False)
    return parser.parse_args(argv)
```

```python

def str2bool(v):
    if v.lower() in ('yes', 'true', 't', 'y', '1'):
        return True
    elif v.lower() in ('no', 'false', 'f', 'n', '0'):
        return False
    else:
        raise argparse.ArgumentTypeError('Boolean value expected')


def create_additional_dataset_eval_kinfacev2(dataset):
    TXT_DESC_ADD_DATASET = [' Child-Young_Fathter', ' Child-Old_Father']
    dataset_array_eval = []
    # Embeddings, Labels, Description
    dataset_child_young = [dataset.dataset_child_young_parents.get_embeddings_np(),
                           dataset.dataset_child_young_parents.get_one_hot_expected_results(),
                           TXT_DESC_ADD_DATASET[0]]
    dataset_array_eval.append(dataset_child_young)
    dataset_child__old = [dataset.dataset_child_old_parents.get_embeddings_np(),
                          dataset.dataset_child_old_parents.get_one_hot_expected_results(),
                          TXT_DESC_ADD_DATASET[1]]
    dataset_array_eval.append(dataset_child__old)
    return dataset_array_eval


def create_data_description(indexes, size_train, size_data):
    TXT_TRAIN = 'Train'
    TXT_TEST = 'Test'
    data_description = []
    for i in range(size_data):
        aux = indexes[i]
        aux_desc = [aux[0], aux[1]]
        if i < size_train:
            aux_desc.append(TXT_TRAIN)
        else:
            aux_desc.append(TXT_TEST)
        data_description.append(aux_desc)
    return data_description


def create_cross_validation_dataset(dataset, n_folds, fold):
    embeds = dataset.all_dataset.get_embeddings_np()
    labels = dataset.all_dataset.get_one_hot_expected_results()
    indexes = dataset.all_dataset.get_indexes()
    # Defines positions on the dataset
```

```
    size_data = embeds.shape[0]
    size_fold = size_data // n_folds
    size_train = size_data - size_fold
    begin = size_fold * fold
    end = begin + size_fold

    # print("Dataset size is {}, n_fold: {}, fold_size: {}, begin: {}, end: {}".format(
    #     size_data, n_folds, size_fold, begin, end))
    # Creates the slices of the dataset
    test_embed = embeds[begin:end]
    test_labels = labels[begin:end]
    test_indexes = indexes[begin:end]

    train_embed = np.concatenate((embeds[:begin], embeds[end:]), axis=0)
    train_labels = np.concatenate((labels[:begin], labels[end:]), axis=0)
    train_indexes = np.concatenate((indexes[:begin], indexes[end:]), axis=0)
    # Rearrange dataset
    embeds_final = np.concatenate((train_embed, test_embed), axis=0)
    labels_final = np.concatenate((train_labels, test_labels), axis=0)
    indexes_final = np.concatenate((train_indexes, test_indexes), axis=0)
    data_description = create_data_description(indexes_final, size_train, size_data)
    add_dataset_array_eval = create_additional_dataset_eval_kinfacev2(dataset)
    # print(len(data_description[size_train:]))
    # print(data_description[size_train:])
    return embeds_final, labels_final, size_train, data_description, add_dataset_array_eval


def generate_model_folder_date(data_dir):
    model_root = os.path.join(data_dir, TXT_MODEL)
    date_str = datetime.datetime.now().strftime("%Y-%m-%d_%H-%M-%S")
    model_root_date = os.path.join(model_root, date_str)
    return model_root_date

def process_results_array(results_array):
    TXT_DESC_RESULTS_ARRAY_SUMM = \
        ['...', 'avg', 'median', 'max', 'index_max', 'min', 'index_min', 'std', 'var']
    results_array_summ = [TXT_DESC_RESULTS_ARRAY_SUMM]
    for item in results_array:
        desc = item[0]
        data = item[1:]
        avg =  np.average(data.astype(np.float64))
        median = np.median(data.astype(np.float64))
        index_max = np.argmax(data.astype(np.float64))
```

```
            max_val = data[index_max].astype(np.float64)
            index_min = np.argmin(data.astype(np.float64))
            min_val = data[index_min].astype(np.float64)
            std = np.std(data.astype(np.float64))
            var = np.var(data.astype(np.float64))
            aux_array = [desc, avg, median, max_val, index_max, min_val,
                index_min, std, var]
            results_array_summ = np.append(results_array_summ, [aux_array],
                axis=0)
        return results_array_summ


def create_hp_array(args, hp_explore_array = None):
    if not args.exploration_mode:
        hp_array = [
            args.learning_rate,
            args.n_epoch,
            args.batch_size,
            args.size_input,
            args.n_classes,
            args.display_step,
            args.keep_prob
        ]
    else:
        hp_array = [
            hp_explore_array[0],
            hp_explore_array[1],
            hp_explore_array[2],
            args.size_input,
            args.n_classes,
            args.display_step,
            hp_explore_array[3]
        ]
    return hp_array

def process_hp_array(hp_array):
    desc_hp_array = [
        'learning_rate',
        'n_epoch',
        'batch_size',
        'size_input',
        'n_classes',
        'display_step',
        'keep_prob',
    ]
    hp_array_summ = np.array([desc_hp_array, hp_array])
    return hp_array_summ
```

```
def train_kinfacev2(args, hp_explore_array = None):
    data_array, size = face_distance_calc.load_data(args.data_dir)
    dataset_kinface = dataset.DatasetKinFaceV2(data_array, size, args.distance_metric)
    hp_array = create_hp_array(args, hp_explore_array)
    # [embed, labels, size_train, data_description]
    # input_data_kinface_model = create_data_kinfacev2(dataset_kinface)
    model_folder_date = generate_model_folder_date(args.data_dir)
    results_array = []
    started = False
    for i in range(args.n_folds):
        model_root_date_iter = os.path.join(model_folder_date, str(i))
        input_data_kinface_model = create_cross_validation_dataset(dataset_kinface, args.n_folds, i)
        results = train_kinship_model(input_data_kinface_model, model_folder_date=model_root_date_iter,
                                      hp_array=hp_array,
                                      # [learning_rate, epoch, batch_size, size_input, n_classes, display_step]
                                      verbose=args.verbose)
        if not started:
            col = [[x] for x in results[:, 0]]
            results_array = col
            started = True
        col = [[x] for x in results[:, 1]]
        results_array = np.append(results_array, col, axis=1)
    results_array_summ = process_results_array(results_array)
    hp_array_summ = process_hp_array(hp_array)
    save_file_npy_csv(model_folder_date, TXT_HP_ARRAY_FILENAME, hp_array_summ)
    save_file_npy_csv(model_folder_date, TXT_RESULTS_ARRAY_SUMM_FILE_NAME, results_array_summ)
    test_acc = results_array_summ[1][1]
    test_acc = float(test_acc)*100
    print("Average accuracy on test is: {:.2f}".format(test_acc))
    model_folder_date_final = create_acc_model_folder_date(model_folder_date, test_acc)
    os.rename(model_folder_date, model_folder_date_final)


def create_acc_model_folder_date(model_folder_date, test_acc):
    pointer = model_folder_date[-1::-1].find("/")
    model_folder_date_final = "{}{:.2f}_{}".format(
        model_folder_date[:-pointer],
        test_acc,
        model_folder_date[-pointer:]
    )
    return model_folder_date_final
```

```python


def explore_mode(args):
    learning_rate_array = [0.03, 0.05, 0.07, 0.09, 0.1]
    n_epoch_array = [6, 10, 14, 18]
    batch_size_array = [10, 40, 80]
    keep_prob_array = [0.4, 0.6, 0.7, 0.8, 0.9, 1.0]
    for learning_rate in learning_rate_array:
        for n_epoch in n_epoch_array:
            for batch_size in batch_size_array:
                for keep_prob in keep_prob_array:
                    hp_explore_array = [learning_rate, n_epoch,
                        batch_size, keep_prob]
                    train_kinfacev2(args, hp_explore_array)


def main(args):
    if args.dataset_type == 0:
        if not args.exploration_mode:
            train_kinfacev2(args)
        else:
            explore_mode(args)



"""ML Model"""
def create_neural_net(x, size_input, n_classes, keep_prob):
    with tf.name_scope('model') as scope:

        # Creates_weights and biases
        layer_1 = tf.layers.dense(
            x, size_input, activation=tf.nn.leaky_relu,
            name='Layer_1')  # pass the first value from iter.get_next()
                as input
        dropout_layer = tf.nn.dropout(layer_1, keep_prob,
                                      name='Dropout_Layer')
        layer_2 = tf.layers.dense(dropout_layer, n_classes, activation=tf
            .nn.leaky_relu,
                                  name='Layer_2')
        prediction = tf.nn.softmax(layer_2, name='prediction')
    return prediction


def create_cost_function(model, y):
    with tf.name_scope('Cost') as scope:
        # Cross entropy loss
        cost_fn = -tf.reduce_sum(y * tf.log(model))
        cost_summ = tf.summary.scalar('Cost_Function', cost_fn)
    return cost_fn, cost_summ
```

```python


def create_optimizer(learning_rate, cost_function):
    with tf.name_scope('train') as scope:
        optimizer = tf.train.GradientDescentOptimizer(learning_rate).minimize(cost_function)
        return optimizer


def create_evaluation_fn(model, y):
    with tf.name_scope('evaluation') as scope:
        """check accuracy"""
        correct_prediction = tf.equal(tf.argmax(model,1), tf.argmax(y,1))
        # accuracy_val, accuracy_fn = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
        accuracy_val, accuracy_fn = \
            tf.metrics.accuracy(labels=tf.argmax(y, 1), predictions=tf.argmax(model, 1))
        roc_val, roc_fn = tf.metrics.auc(tf.argmax(y, 1), tf.argmax(model, 1))
        precision_val, precision_fn = tf.metrics.precision(tf.argmax(y, 1), tf.argmax(model, 1))
        recall_val, recall_fn = tf.metrics.recall(tf.argmax(y, 1), tf.argmax(model, 1))
        # Summary
        accuracy_summ = tf.summary.scalar('Accuracy', accuracy_val)
        roc_score_summ = tf.summary.scalar('ROC', roc_val)
        precision_val_summ = tf.summary.scalar('Precision', precision_val)
        recall_val_summ = tf.summary.scalar('Recall', recall_val)
        #To understand this order see TXT_ACC_ARRAY on function: process_evaluation
        values = [accuracy_val, roc_val, precision_val, recall_val]
        functions = [accuracy_fn, roc_fn, precision_fn, recall_fn]
        summaries = []
    return values, functions


def create_positive_false_fn(model, y):
    with tf.name_scope('positive_false_numbers') as scope:
        true_p_n, true_p_fn = tf.metrics.true_positives(tf.argmax(y, 1), tf.argmax(model, 1))
        false_p_n, false_p_fn = tf.metrics.false_negatives(tf.argmax(y, 1), tf.argmax(model, 1))
        true_n_n, true_n_fn = tf.metrics.true_negatives(tf.argmax(y, 1), tf.argmax(model, 1))
        false_n_n, false_n_fn = tf.metrics.false_negatives(tf.argmax(y, 1), tf.argmax(model, 1))
        # Summary
```

```python
        true_p_n_summ = tf.summary.scalar('True_Positive', true_p_n)
        false_p_n_summ = tf.summary.scalar('False_Positive', false_p_n)
        true_n_n_summ = tf.summary.scalar('True_Negative', true_n_n)
        false_n_n_summ = tf.summary.scalar('False_Negative', false_n_n)
        # To understand this order see TXT_POSITIVE_FALSE on function: process_positive_false
        quantities = [true_p_n, false_p_n, true_n_n, false_n_n]
        functions = [true_p_fn, false_p_fn, true_n_fn, false_n_fn]
        return quantities, functions


def process_positive_false(positive_false_result, size, suffix, verbose=False):
    """To change the order of the itens on the array is necessary to change the variable: TXT_POSITIVE_FALSE"""
    TXT_POSITIVE_FALSE = ['True positive', 'False positive', 'True negative', 'False negative']
    TXT_SIZE = 'Size of the sample'
    # Specific statistics name
    positive_false_array = [[TXT_SIZE + suffix, size]]
    if verbose:
        print('{}: {}'.format(TXT_SIZE, size), end=' | ')
    for i in range(len(positive_false_result)):
        desc = TXT_POSITIVE_FALSE[i] + suffix
        value = positive_false_result[i]
        if verbose:
            print('{}: {}'.format(desc, value), end = ' | ')
        positive_false_array.append([desc, value])
    if verbose:
        print()
    return positive_false_array


def process_evaluation(evaluation_value_array, suffix, verbose=False):
    TXT_EVAL_ARRAY = ['Accuracy', 'ROC', 'Precision', 'Recall']
    eval_array = []
    for i in range(len(TXT_EVAL_ARRAY)):
        desc = TXT_EVAL_ARRAY[i] + suffix
        value = evaluation_value_array[i]
        if verbose:
            print('{}: {}'.format(desc, value), end=' | ')
        eval_array.append([desc, value])
    if verbose:
        print()
    return eval_array


def create_histogram_variables():
    with tf.variable_scope("Layer_1", reuse=True):
```

```
            weights = tf.get_variable('kernel')
            bias = tf.get_variable('bias')
            w_1 = tf.summary.histogram("Weights_1", weights)
            b_1 = tf.summary.histogram("Biases_1", bias)
        with tf.variable_scope('Layer_2', reuse=True):
            weights_2 = tf.get_variable('kernel')
            bias_2 = tf.get_variable('bias')
            w_2 = tf.summary.histogram("Weights_2", weights_2)
            b_2 = tf.summary.histogram("Biases_2", bias_2)
        return [w_1, b_1, w_2, b_2]


def create_txt_summary(name, value):
    if (value - int(value)) != 0:
        txt = '{:.10f}'.format(value)
    else:
        txt = '{}'.format(int(value))
    summary_op = tf.summary.text(name, tf.convert_to_tensor(txt))
    return summary_op


def create_array_feed_eval(dataset_array, x, y, batch_size, keep_prob):
    array_feed = []
    for dataset in dataset_array:
        aux_arr_feed = [{x: dataset[0], y: dataset[1], batch_size:
            dataset[0].shape[0], keep_prob: 1.0}, dataset[2]]
        array_feed.append(aux_arr_feed)
    return array_feed


def train_kinship_model(input_data, model_folder_date,
                        hp_array = [0.07, 10, 10, 128, 2, 1, 0.4],
                        verbose=False, store=True,
                        calc_positive_false = False):
    """hp_array is formed by [learning_rate, epoch, batch_size,
        size_input, n_classes, display_step, keep_prob_val]
    """
    LOG_DIR_NAME = 'logs'
    MODEL_DIR_NAME = 'model'
    DATA_DESC_NAME = 'data_desc.npy'
    RESULTS_SUMM_NAME = 'results_sum.npy'

    # Separates data from input_data
    embeds = input_data[0]
    labels = input_data[1]
    size_train = input_data[2]
    data_description = input_data[3]
    add_dataset_array_eval = input_data[4]

```

```python

    #Separates hyper Parameters
    hp_learning_rate = hp_array[0]
    hp_epoch = hp_array[1]
    hp_batch_size = hp_array[2]
    hp_size_input = hp_array[3]
    hp_n_classes = hp_array[4]
    hp_display_step = hp_array[5]

    keep_prob_val = hp_array[6]

    #Creates
    log_dir = os.path.join(model_folder_date, LOG_DIR_NAME)
    model_dir = os.path.join(model_folder_date, MODEL_DIR_NAME)
    os.makedirs(model_dir)
    model_path_name = os.path.join(model_dir, MODEL_DIR_NAME)

    # TF graph input
    batch_size = tf.placeholder(tf.int64, name='batch_size')
    x = tf.placeholder(tf.float32, [None, hp_size_input], name='x')  # mnist data image
    y = tf.placeholder(tf.float32, [None, hp_n_classes], name='y')
    keep_prob = tf.placeholder(tf.float32, name='keep_prob')
    dataset_1 = tf.data.Dataset.from_tensor_slices((x, y)).batch(batch_size).repeat()
    keep_prob_train_dict = {keep_prob: keep_prob_val}
    keep_prob_test_dict = {keep_prob: 1.0}

    train_data = (embeds[:size_train], labels[:size_train])
    test_data = (embeds[size_train:], labels[size_train:])

    iterator = dataset_1.make_initializable_iterator()
    feat, lbl = iterator.get_next()

    # Create a model
    model = create_neural_net(feat, hp_size_input, hp_n_classes, keep_prob)  # Softmax

    cost_fn, cost_summ = create_cost_function(model, lbl)
    if store:
        train_summ_op_hist = create_histogram_variables()

    # gradient descent
    optimizer = create_optimizer(hp_learning_rate, cost_fn)

    evaluation_values_array, evaluation_fn_array = create_evaluation_fn(model, lbl)
    if calc_positive_false:
        positive_false_n_array, positive_false_fn_array =
```

```
            create_positive_false_fn(model, lbl)

    # Initialize variables
    init = tf.group(tf.global_variables_initializer(), tf.
        local_variables_initializer())

    saver = tf.train.Saver()

    # Launch graph
    with tf.Session() as sess:
        sess.run(init)
        feed_train = {x: train_data[0], y: train_data[1], batch_size:
            hp_batch_size}
        sess.run(iterator.initializer, feed_dict=feed_train)
        if store:
            summary_writer = tf.summary.FileWriter(
                logdir=log_dir,
                graph=sess.graph)
        for i in range(hp_epoch):
            total_batch = int(size_train / hp_batch_size)
            # Process all batches
            for j in range(total_batch):
                _, cost = sess.run([optimizer, cost_fn], feed_dict=
                    keep_prob_train_dict)
                # Calculate logs
                index = i * total_batch + j
                if store:
                    summary_hist = sess.run(train_summ_op_hist, feed_dict
                        =keep_prob_train_dict)
                    summary_cost = sess.run(cost_summ, feed_dict=
                        keep_prob_train_dict)
                    summary_writer.add_summary(summary_cost, index)
                    # Write histogram logs for each interation
                    for item in summary_hist:
                        summary_writer.add_summary(item, index)
                # Shows status on display_step
            if verbose and i % hp_display_step == 0:
                print('Interation: {:04d}  | Cost: {} '.format(i + 1,
                    cost))
        if verbose:
            print('Training completed!')
        """Builds the dictionaries to process evaluation"""
        feed_dataset_eval = [[test_data[0], test_data[1], TXT_TEST_SUFFIX
            ],
                             [train_data[0], train_data[1],
                                 TXT_TRAIN_SUFFIX]]
        feed_dataset_eval.extend(add_dataset_array_eval)
        feed_array_eval = create_array_feed_eval(feed_dataset_eval, x, y,
            batch_size, keep_prob)
```

```
        """Starts Evaluation"""
        if verbose:
            print('Starting evaluation on deisgned datasets ')
        results = []
        for feed in feed_array_eval:
            aux_feed = feed[0]
            suffix = feed[1]
            size = aux_feed[batch_size]
            if verbose:
                print('Starting feed:{}'.format(suffix))
            sess.run(iterator.initializer, feed_dict=aux_feed)
            # Run graph operations
            evaluation_array = sess.run(evaluation_fn_array, feed_dict=keep_prob_test_dict)
            eval_array_ident = process_evaluation(evaluation_array, suffix, verbose)
            results.extend(eval_array_ident)
            if calc_positive_false:
                positive_false_array = sess.run(positive_false_fn_array, feed_dict=keep_prob_test_dict)
                pos_false_array_ident = process_positive_false(positive_false_array, size, suffix, verbose)
                results.extend(pos_false_array_ident)
        if store:
            for result in results:
                summary_op = create_txt_summary(result[0], result[1])
                txt_summ = sess.run(summary_op, feed_dict=keep_prob_test_dict)
                summary_writer.add_summary(txt_summ, 0)
        results = np.array(results)
        # Saves model
        if store:
            saver.save(sess, model_path_name, write_meta_graph=True)
            # Saves description of data used
            save_file_npy_csv(log_dir, DATA_DESC_NAME, data_description) # Save data desc
            # Saves the results
            save_file_npy_csv(log_dir, RESULTS_SUMM_NAME, results)
            print('Data saved on: {}'.format(model_folder_date))
            summary_writer.close()
    tf.reset_default_graph()
    return results

if __name__ == '__main__':
    main(parse_arguments(sys.argv[1:]))
```

List 6.2: Complete FaceNet model source code with Inception-ResNet-v1

```
# Copyright 2016 The TensorFlow Authors. All Rights Reserved.
#
```

```
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
# ==============================================================================

"""Contains the definition of the Inception Resnet V1 architecture.
As described in http://arxiv.org/abs/1602.07261.
  Inception-v4, Inception-ResNet and the Impact of Residual Connections
    on Learning
  Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, Alex Alemi
"""
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import tensorflow as tf
import tensorflow.contrib.slim as slim

# Inception-Resnet-A
def block35(net, scale=1.0, activation_fn=tf.nn.relu, scope=None, reuse=None):
    """Builds the 35x35 resnet block."""
    with tf.variable_scope(scope, 'Block35', [net], reuse=reuse):
        with tf.variable_scope('Branch_0'):
            tower_conv = slim.conv2d(net, 32, 1, scope='Conv2d_1x1')
        with tf.variable_scope('Branch_1'):
            tower_conv1_0 = slim.conv2d(net, 32, 1, scope='Conv2d_0a_1x1')
            tower_conv1_1 = slim.conv2d(tower_conv1_0, 32, 3, scope='Conv2d_0b_3x3')
        with tf.variable_scope('Branch_2'):
            tower_conv2_0 = slim.conv2d(net, 32, 1, scope='Conv2d_0a_1x1')
            tower_conv2_1 = slim.conv2d(tower_conv2_0, 32, 3, scope='Conv2d_0b_3x3')
            tower_conv2_2 = slim.conv2d(tower_conv2_1, 32, 3, scope='Conv2d_0c_3x3')
        mixed = tf.concat([tower_conv, tower_conv1_1, tower_conv2_2], 3)
```

```
        up = slim.conv2d(mixed, net.get_shape()[3], 1, normalizer_fn=None,
                         activation_fn=None, scope='Conv2d_1x1')
        net += scale * up
        if activation_fn:
            net = activation_fn(net)
    return net

# Inception-Resnet-B
def block17(net, scale=1.0, activation_fn=tf.nn.relu, scope=None, reuse=None):
    """Builds the 17x17 resnet block."""
    with tf.variable_scope(scope, 'Block17', [net], reuse=reuse):
        with tf.variable_scope('Branch_0'):
            tower_conv = slim.conv2d(net, 128, 1, scope='Conv2d_1x1')
        with tf.variable_scope('Branch_1'):
            tower_conv1_0 = slim.conv2d(net, 128, 1, scope='Conv2d_0a_1x1')
            tower_conv1_1 = slim.conv2d(tower_conv1_0, 128, [1, 7],
                                        scope='Conv2d_0b_1x7')
            tower_conv1_2 = slim.conv2d(tower_conv1_1, 128, [7, 1],
                                        scope='Conv2d_0c_7x1')
        mixed = tf.concat([tower_conv, tower_conv1_2], 3)
        up = slim.conv2d(mixed, net.get_shape()[3], 1, normalizer_fn=None,
                         activation_fn=None, scope='Conv2d_1x1')
        net += scale * up
        if activation_fn:
            net = activation_fn(net)
    return net


# Inception-Resnet-C
def block8(net, scale=1.0, activation_fn=tf.nn.relu, scope=None, reuse=None):
    """Builds the 8x8 resnet block."""
    with tf.variable_scope(scope, 'Block8', [net], reuse=reuse):
        with tf.variable_scope('Branch_0'):
            tower_conv = slim.conv2d(net, 192, 1, scope='Conv2d_1x1')
        with tf.variable_scope('Branch_1'):
            tower_conv1_0 = slim.conv2d(net, 192, 1, scope='Conv2d_0a_1x1')
            tower_conv1_1 = slim.conv2d(tower_conv1_0, 192, [1, 3],
                                        scope='Conv2d_0b_1x3')
            tower_conv1_2 = slim.conv2d(tower_conv1_1, 192, [3, 1],
                                        scope='Conv2d_0c_3x1')
        mixed = tf.concat([tower_conv, tower_conv1_2], 3)
        up = slim.conv2d(mixed, net.get_shape()[3], 1, normalizer_fn=None,
```

```python
                          activation_fn=None, scope='Conv2d_1x1')
        net += scale * up
        if activation_fn:
            net = activation_fn(net)
    return net

def reduction_a(net, k, l, m, n):
    with tf.variable_scope('Branch_0'):
        tower_conv = slim.conv2d(net, n, 3, stride=2, padding='VALID',
                                 scope='Conv2d_1a_3x3')
    with tf.variable_scope('Branch_1'):
        tower_conv1_0 = slim.conv2d(net, k, 1, scope='Conv2d_0a_1x1')
        tower_conv1_1 = slim.conv2d(tower_conv1_0, l, 3,
                                    scope='Conv2d_0b_3x3')
        tower_conv1_2 = slim.conv2d(tower_conv1_1, m, 3,
                                    stride=2, padding='VALID',
                                    scope='Conv2d_1a_3x3')
    with tf.variable_scope('Branch_2'):
        tower_pool = slim.max_pool2d(net, 3, stride=2, padding='VALID',
                                     scope='MaxPool_1a_3x3')
    net = tf.concat([tower_conv, tower_conv1_2, tower_pool], 3)
    return net

def reduction_b(net):
    with tf.variable_scope('Branch_0'):
        tower_conv = slim.conv2d(net, 256, 1, scope='Conv2d_0a_1x1')
        tower_conv_1 = slim.conv2d(tower_conv, 384, 3, stride=2,
                                   padding='VALID', scope='Conv2d_1a_3x3'
                                   )
    with tf.variable_scope('Branch_1'):
        tower_conv1 = slim.conv2d(net, 256, 1, scope='Conv2d_0a_1x1')
        tower_conv1_1 = slim.conv2d(tower_conv1, 256, 3, stride=2,
                                    padding='VALID', scope='Conv2d_1a_3x3
                                    ')
    with tf.variable_scope('Branch_2'):
        tower_conv2 = slim.conv2d(net, 256, 1, scope='Conv2d_0a_1x1')
        tower_conv2_1 = slim.conv2d(tower_conv2, 256, 3,
                                    scope='Conv2d_0b_3x3')
        tower_conv2_2 = slim.conv2d(tower_conv2_1, 256, 3, stride=2,
                                    padding='VALID', scope='Conv2d_1a_3x3
                                    ')
    with tf.variable_scope('Branch_3'):
        tower_pool = slim.max_pool2d(net, 3, stride=2, padding='VALID',
                                     scope='MaxPool_1a_3x3')
    net = tf.concat([tower_conv_1, tower_conv1_1,
                     tower_conv2_2, tower_pool], 3)
    return net

def inference(images, keep_probability, phase_train=True,
```

```
            bottleneck_layer_size=128, weight_decay=0.0, reuse=None):
    batch_norm_params = {
        # Decay for the moving averages.
        'decay': 0.995,
        # epsilon to prevent 0s in variance.
        'epsilon': 0.001,
        # force in-place updates of mean and variance estimates
        'updates_collections': None,
        # Moving averages ends up in the trainable variables collection
        'variables_collections': [ tf.GraphKeys.TRAINABLE_VARIABLES ],
    }

    with slim.arg_scope([slim.conv2d, slim.fully_connected],
                        weights_initializer=slim.initializers.xavier_initializer(),
                        weights_regularizer=slim.l2_regularizer(weight_decay),
                        normalizer_fn=slim.batch_norm,
                        normalizer_params=batch_norm_params):
        return inception_resnet_v1(images, is_training=phase_train,
              dropout_keep_prob=keep_probability, bottleneck_layer_size=bottleneck_layer_size, reuse=reuse)


def inception_resnet_v1(inputs, is_training=True,
                        dropout_keep_prob=0.8,
                        bottleneck_layer_size=128,
                        reuse=None,
                        scope='InceptionResnetV1'):
    """Creates the Inception Resnet V1 model.
    Args:
      inputs: a 4-D tensor of size [batch_size, height, width, 3].
      num_classes: number of predicted classes.
      is_training: whether is training or not.
      dropout_keep_prob: float, the fraction to keep before final layer.
      reuse: whether or not the network and its variables should be reused. To be
        able to reuse 'scope' must be given.
      scope: Optional variable_scope.
    Returns:
      logits: the logits outputs of the model.
      end_points: the set of end_points from the inception model.
    """
    end_points = {}

    with tf.variable_scope(scope, 'InceptionResnetV1', [inputs], reuse=reuse):
        with slim.arg_scope([slim.batch_norm, slim.dropout],
                            is_training=is_training):
```

```
        with slim.arg_scope([slim.conv2d, slim.max_pool2d, slim.avg_pool2d],
                            stride=1, padding='SAME'):

            # 149 x 149 x 32
            net = slim.conv2d(inputs, 32, 3, stride=2, padding='VALID',
                              scope='Conv2d_1a_3x3')
            end_points['Conv2d_1a_3x3'] = net
            # 147 x 147 x 32
            net = slim.conv2d(net, 32, 3, padding='VALID',
                              scope='Conv2d_2a_3x3')
            end_points['Conv2d_2a_3x3'] = net
            # 147 x 147 x 64
            net = slim.conv2d(net, 64, 3, scope='Conv2d_2b_3x3')
            end_points['Conv2d_2b_3x3'] = net
            # 73 x 73 x 64
            net = slim.max_pool2d(net, 3, stride=2, padding='VALID',
                                  scope='MaxPool_3a_3x3')
            end_points['MaxPool_3a_3x3'] = net
            # 73 x 73 x 80
            net = slim.conv2d(net, 80, 1, padding='VALID',
                              scope='Conv2d_3b_1x1')
            end_points['Conv2d_3b_1x1'] = net
            # 71 x 71 x 192
            net = slim.conv2d(net, 192, 3, padding='VALID',
                              scope='Conv2d_4a_3x3')
            end_points['Conv2d_4a_3x3'] = net
            # 35 x 35 x 256
            net = slim.conv2d(net, 256, 3, stride=2, padding='VALID',
                              scope='Conv2d_4b_3x3')
            end_points['Conv2d_4b_3x3'] = net

            # 5 x Inception-resnet-A
            net = slim.repeat(net, 5, block35, scale=0.17)
            end_points['Mixed_5a'] = net

            # Reduction-A
            with tf.variable_scope('Mixed_6a'):
                net = reduction_a(net, 192, 192, 256, 384)
            end_points['Mixed_6a'] = net

            # 10 x Inception-Resnet-B
            net = slim.repeat(net, 10, block17, scale=0.10)
            end_points['Mixed_6b'] = net

            # Reduction-B
            with tf.variable_scope('Mixed_7a'):
                net = reduction_b(net)
```

```
            end_points['Mixed_7a'] = net

            # 5 x Inception-Resnet-C
            net = slim.repeat(net, 5, block8, scale=0.20)
            end_points['Mixed_8a'] = net

            net = block8(net, activation_fn=None)
            end_points['Mixed_8b'] = net

            with tf.variable_scope('Logits'):
                end_points['PrePool'] = net
                #pylint: disable=no-member
                net = slim.avg_pool2d(net, net.get_shape()[1:3],
                    padding='VALID',
                                      scope='AvgPool_1a_8x8')
                net = slim.flatten(net)

                net = slim.dropout(net, dropout_keep_prob,
                    is_training=is_training,
                                   scope='Dropout')

                end_points['PreLogitsFlatten'] = net

            net = slim.fully_connected(net, bottleneck_layer_size,
                activation_fn=None,
                    scope='Bottleneck', reuse=False)

    return net, end_points
```

List 6.3: Reduction-A code from FaceNet on Table 4.2

```
def reduction_a(net, k, l, m, n):
    with tf.variable_scope('Branch_0'):
        tower_conv = slim.conv2d(net, n, 3, stride=2, padding='VALID',
                                 scope='Conv2d_1a_3x3')
    with tf.variable_scope('Branch_1'):
        tower_conv1_0 = slim.conv2d(net, k, 1, scope='Conv2d_0a_1x1')
        tower_conv1_1 = slim.conv2d(tower_conv1_0, l, 3,
                                    scope='Conv2d_0b_3x3')
        tower_conv1_2 = slim.conv2d(tower_conv1_1, m, 3,
                                    stride=2, padding='VALID',
                                    scope='Conv2d_1a_3x3')
    with tf.variable_scope('Branch_2'):
        tower_pool = slim.max_pool2d(net, 3, stride=2, padding='VALID',
                                     scope='MaxPool_1a_3x3')
    net = tf.concat([tower_conv, tower_conv1_2, tower_pool], 3)
    return net
```

List 6.4: Reduction-B code from FaceNet on Table 4.2

```
def reduction_a(net, k, l, m, n):
```

```python
    with tf.variable_scope('Branch_0'):
        tower_conv = slim.conv2d(net, n, 3, stride=2, padding='VALID',
                                 scope='Conv2d_1a_3x3')
    with tf.variable_scope('Branch_1'):
        tower_conv1_0 = slim.conv2d(net, k, 1, scope='Conv2d_0a_1x1')
        tower_conv1_1 = slim.conv2d(tower_conv1_0, l, 3,
                                    scope='Conv2d_0b_3x3')
        tower_conv1_2 = slim.conv2d(tower_conv1_1, m, 3,
                                    stride=2, padding='VALID',
                                    scope='Conv2d_1a_3x3')
    with tf.variable_scope('Branch_2'):
        tower_pool = slim.max_pool2d(net, 3, stride=2, padding='VALID',
                                     scope='MaxPool_1a_3x3')
    net = tf.concat([tower_conv, tower_conv1_2, tower_pool], 3)
    return net
```

List 6.5: Function that generates the model

```python
import tensorflow as tf
"""ML Model"""
def create_neural_net(x, size_input, n_classes, keep_prob):
    with tf.name_scope('model') as scope:

        # Creates_weights and biases
        layer_1 = tf.layers.dense(
            x, size_input, activation=tf.nn.leaky_relu,
            name='Layer_1')  # pass the first value from iter.get_next() as input
        dropout_layer = tf.nn.dropout(layer_1, keep_prob,
                                      name='Dropout_Layer')
        layer_2 = tf.layers.dense(dropout_layer, n_classes, activation=tf.nn.leaky_relu,
                                  name='Layer_2')
        prediction = tf.nn.softmax(layer_2, name='prediction')
    return prediction
```

List 6.6: Cost function for the linear model

```python
def create_cost_function(model, y):
    with tf.name_scope('Cost') as scope:
        # Cross entropy loss
        cost_fn = -tf.reduce_sum(y * tf.log(model))
        cost_summ = tf.summary.scalar('Cost_Function', cost_fn)
    return cost_fn, cost_summ
```

List 6.7: Optimizer for the linear model

```python
def create_optimizer(learning_rate, cost_function):
    with tf.name_scope('train') as scope:
        optimizer = tf.train.GradientDescentOptimizer(learning_rate).minimize(cost_function)
```

```
        return optimizer
```

List 6.8: Script to process dataset images

```
python3 facenet/src/align/align_dataset_mtcnn.py \
origin_folder \ # Original images from dataset
output_folder \ # Aligned facial images resized to 160x160
--image_size 160 \
--margin 0 \
--random_order \
--gpu_memory_fraction 0.7
```

List 6.9: Script to create grayscale images

```
python3 facenet/process_datasets/KinFaceW/convert_gray_scale.py \
UB_Kinface_mtcnnpy_160_0 \
UB_Kinface_mtcnnpy_160_0_pb
```

List 6.10: Code that generates grayscale image(convertgrayscale.py)s

```
import os
import sys
import argparse
import cv2

def process_save_pb(src_img_path, out_img_path):
    img = cv2.imread(src_img_path)
    img_gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    cv2.imwrite(out_img_path, img_gray)

def process_dataset(args):
    folder_list = [f for f in os.listdir(args.src_dir) if os.path.isdir(
        os.path.join(args.src_dir, f))]
    for f in folder_list:
        print('Processing folder {}...'.format(f))
        src_folder = os.path.join(args.src_dir, f)
        src_file_list = [f for f in os.listdir(src_folder) if os.path.
            isfile(os.path.join(src_folder, f))]
        out_folder = os.path.join(args.out_dir, f)
        if not os.path.exists(out_folder):
            os.makedirs(out_folder)
        for img_path in src_file_list:
            src_img_path = os.path.join(src_folder, img_path)
            out_img_path = os.path.join(out_folder, img_path)
            process_save_pb(src_img_path, out_img_path)



def parse_arguments(argv):
    parser = argparse.ArgumentParser()
    parser.add_argument('src_dir', type=str,
```

```
                          help='Enter the dir where the dataset is')
    parser.add_argument('out_dir', type=str,
                          help='Enter the dir where the dataset in gray
                             scale is going to be saved')
    return parser.parse_args(argv)

def main(args):
    process_dataset(args)

if __name__ == '__main__':
    main(parse_arguments(sys.argv[1:]))
```

List 6.11: Script that trains FaceNet

```
python3 facenet/src/train_softmax.py \
--logs_base_dir facenet/trained/vgg2_mtcnn_160_0_2/logs/20180530-033902 \
--models_base_dir facenet/trained/vgg2_mtcnn_160_0_2/models
    /20180530-033902 \
--data_dir MTCNN_Aligned/facenet_160_0/vggface2_train_160_0/ \
--lfw_dir MTCNN_Aligned/facenet_160_0/lfw_160_0/ \
--image_size 160 \
--model_def models.inception_resnet_v1 \
--optimizer ADAM \
--learning_rate -1 \
--max_nrof_epochs 500 \
--batch_size 40 \
--keep_probability 0.4 \
--use_fixed_image_standardization \
--learning_rate_schedule_file facenet/data/
    learning_rate_schedule_classifier_vggface2.txt \
--weight_decay 5e-4 \
--embedding_size 128 \
--lfw_distance_metric 0 \
--validation_set_split_ratio 0.01 \
--validate_every_n_epochs 5 \
--gpu_memory_fraction 0.8
```

List 6.12: Functions that generate the cross-validation dataset for each cycle

```
import numpy as np
def create_data_description(indexes, size_train, size_data):
    TXT_TRAIN = 'Train'
    TXT_TEST = 'Test'
    data_description = []
    for i in range(size_data):
        aux = indexes[i]
        aux_desc = [aux[0], aux[1]]
        if i < size_train:
            aux_desc.append(TXT_TRAIN)
        else:
            aux_desc.append(TXT_TEST)
```

```python
            data_description.append(aux_desc)
    return data_description


def create_cross_validation_dataset(dataset, n_folds, fold):
    embeds = dataset.all_dataset.get_embeddings_np()
    labels = dataset.all_dataset.get_one_hot_expected_results()
    indexes = dataset.all_dataset.get_indexes()
    # Defines positions on the dataset
    size_data = embeds.shape[0]
    size_fold = size_data // n_folds
    size_train = size_data - size_fold
    begin = size_fold * fold
    end = begin + size_fold

    # print("Dataset size is {}, n_fold: {}, fold_size: {}, begin: {}, end: {}".format(
    #       size_data, n_folds, size_fold, begin, end))
    # Creates the slices of the dataset
    test_embed = embeds[begin:end]
    test_labels = labels[begin:end]
    test_indexes = indexes[begin:end]

    train_embed = np.concatenate((embeds[:begin], embeds[end:]), axis=0)
    train_labels = np.concatenate((labels[:begin], labels[end:]), axis=0)
    train_indexes = np.concatenate((indexes[:begin], indexes[end:]), axis=0)
    # Rearrange dataset
    embeds_final = np.concatenate((train_embed, test_embed), axis=0)
    labels_final = np.concatenate((train_labels, test_labels), axis=0)
    indexes_final = np.concatenate((train_indexes, test_indexes), axis=0)
    data_description = create_data_description(indexes_final, size_train, size_data)
    add_dataset_array_eval = create_additional_dataset_eval_kinfacev2(dataset)
    # print(len(data_description[size_train:]))
    # print(data_description[size_train:])
    return embeds_final, labels_final, size_train, data_description, add_dataset_array_eval
```

List 6.13: Inception-A source code for Inception-ResNet-v1

```python
import tensorflow as tf
# Inception-Resnet-A
def block35(net, scale=1.0, activation_fn=tf.nn.relu, scope=None, reuse=None):
    """Builds the 35x35 resnet block."""
    with tf.variable_scope(scope, 'Block35', [net], reuse=reuse):
        with tf.variable_scope('Branch_0'):
            tower_conv = slim.conv2d(net, 32, 1, scope='Conv2d_1x1')
```

```
        with tf.variable_scope('Branch_1'):
            tower_conv1_0 = slim.conv2d(net, 32, 1, scope='Conv2d_0a_1x1')
            tower_conv1_1 = slim.conv2d(tower_conv1_0, 32, 3, scope='Conv2d_0b_3x3')
        with tf.variable_scope('Branch_2'):
            tower_conv2_0 = slim.conv2d(net, 32, 1, scope='Conv2d_0a_1x1')
            tower_conv2_1 = slim.conv2d(tower_conv2_0, 32, 3, scope='Conv2d_0b_3x3')
            tower_conv2_2 = slim.conv2d(tower_conv2_1, 32, 3, scope='Conv2d_0c_3x3')
        mixed = tf.concat([tower_conv, tower_conv1_1, tower_conv2_2], 3)
        up = slim.conv2d(mixed, net.get_shape()[3], 1, normalizer_fn=None,
                         activation_fn=None, scope='Conv2d_1x1')
        net += scale * up
        if activation_fn:
            net = activation_fn(net)
    return net
```

List 6.14: Inception-B source code for Inception-ResNet-v1

```
import tensorflow as tf
# Inception-Resnet-B
def block17(net, scale=1.0, activation_fn=tf.nn.relu, scope=None, reuse=None):
    """Builds the 17x17 resnet block."""
    with tf.variable_scope(scope, 'Block17', [net], reuse=reuse):
        with tf.variable_scope('Branch_0'):
            tower_conv = slim.conv2d(net, 128, 1, scope='Conv2d_1x1')
        with tf.variable_scope('Branch_1'):
            tower_conv1_0 = slim.conv2d(net, 128, 1, scope='Conv2d_0a_1x1')
            tower_conv1_1 = slim.conv2d(tower_conv1_0, 128, [1, 7],
                                        scope='Conv2d_0b_1x7')
            tower_conv1_2 = slim.conv2d(tower_conv1_1, 128, [7, 1],
                                        scope='Conv2d_0c_7x1')
        mixed = tf.concat([tower_conv, tower_conv1_2], 3)
        up = slim.conv2d(mixed, net.get_shape()[3], 1, normalizer_fn=None,
                         activation_fn=None, scope='Conv2d_1x1')
        net += scale * up
        if activation_fn:
            net = activation_fn(net)
    return net
```

List 6.15: Inception-C source code for Inception-ResNet-v1

```
import tensorflow as tf
# Inception-Resnet-C
```

```python
def block8(net, scale=1.0, activation_fn=tf.nn.relu, scope=None, reuse=None):
    """Builds the 8x8 resnet block."""
    with tf.variable_scope(scope, 'Block8', [net], reuse=reuse):
        with tf.variable_scope('Branch_0'):
            tower_conv = slim.conv2d(net, 192, 1, scope='Conv2d_1x1')
        with tf.variable_scope('Branch_1'):
            tower_conv1_0 = slim.conv2d(net, 192, 1, scope='Conv2d_0a_1x1')
            tower_conv1_1 = slim.conv2d(tower_conv1_0, 192, [1, 3],
                                        scope='Conv2d_0b_1x3')
            tower_conv1_2 = slim.conv2d(tower_conv1_1, 192, [3, 1],
                                        scope='Conv2d_0c_3x1')
        mixed = tf.concat([tower_conv, tower_conv1_2], 3)
        up = slim.conv2d(mixed, net.get_shape()[3], 1, normalizer_fn=None,
                         activation_fn=None, scope='Conv2d_1x1')
        net += scale * up
        if activation_fn:
            net = activation_fn(net)
    return net
```